DOCUMENT RESUME

ED 262 763 IR 011 841

TITLE	LOGO in the Classroom.
INSTITUTION	International Council for Computers in Education, Eugene, Oreg.
PUB DATE	85
NOTE	85p.; For the 20-session course, Logo in the Classroom, see ED 248 847.
AVAILABLE FROM	International Council for Computers in Education, University of Oregon, 1787 Agate Street, Eugene, OR 97403.
PUB TYPE	Guides - Classroom Use - Guides (For Teachers) (052) -- Computer Programs (101) -- Journal Articles (080)
JOURNAL CIT	The Computing Teacher; Sep 1983-Mar 1985
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	Computer Graphics; *Computer Software; Educational Strategies; Elementary Secondary Education; Instructional Materials; *Microcomputers; Programing; *Programing Languages; Resource Units; Teaching Methods
IDENTIFIERS	*Computer Uses in Education; *LOGO Programing Language

ABSTRACT

This collection of articles and features reprinted from "The Computing Teacher" begins with an outline (the table of contents) of a 20-session course, "Logo in the Classroom," by Shirley Torgerson, Mary Kay Kriley, and Janet Stone. The text of the first session and a student exercise sheet are also provided. Five articles included in the collection are (1) "Logo Today: Vision or Reality" (Uri Leron); (2) "Animation with MIT Logo" (Nancy Parks Sopp); (3) "Helping Students with Recursion: Teaching Strategies" (Tim Riordan); (4) "Turtle Fever" (Rick Billstein); and (5) "Putting Logo to Work" (Hillel Weintraub). Copies of a regular feature, "The Logo Center," are also included. These columns by Kathleen Martin and Tim Riordan address such topics as spirolaterals, adventure stories, tangrams and turtles, databases, polyspirals, sprites, thinking in "chunks," non-graphics applications, and using older students to help teach Logo to younger children. One guest column by Tim Barclay focuses on teaching mathematics with MIT Logo. (JB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

LOGO IN THE CLASSROOM

The following table of contents and session is from Logo in the Classroom by Shirley Torgerson, Mary Kay Kriley and Janet Stone.

Copyright © ICCE 198

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
This document has been reproduced as received from the person or organization originating it
Minor changes have been made to improve reproduction quality
Points of view or opinions stated in this document do not necessarily represent official NIE position or policy

ED262763

Table of Contents

Preface 4

Introduction 6
What Is Logo? 6
How Does Logo Fit into the Curriculum? 7
Why "Play Turtle?" 7
How Do I Get Started? 7
What Do I Need to Know About? 10

Scope and Sequence Chart 12

Session 1 13
To introduce students to the Logo primitives FORWARD, BACK, RIGHT and LEFT through self-involvement in spatial relationship activities.
To reinforce following and giving precise directions.

Session 2 17
To introduce abbreviated forms for the Logo primitives.
To provide practice in using the Logo primitives as spatial commands through self-involved activities.

Session 3 21
To introduce students to the computer and to the Logo Turtle.

Session 4 25
To examine how we change directions by turning, or pivoting.
To introduce the protractor as a tool for measuring the amount of turning.

Session 5 32
To introduce the concept of writing a set of directions.
To reinforce the concept of measuring the amount of turning.
To provide practice in using the protractor.

Session 6 35
To reinforce measuring "the amount of turning" in an angle.
To review metric measurement relative to Turtle steps.

Session 7 38
To introduce and give practice in using a small, desk-size protractor.
To introduce the concept of "angle" and relate it to the amount of turning in changing directions.

Session 8 45
To introduce the Logo primitives PENUP and PENDOWN, and the primitive for erasing.
To help students develop concepts of distance in Turtle steps and in angles by the amount of turning.
To provide practice for internalizing the Logo primitives.

IR011841

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY
David Morsund

© Copyright 1985 ICCE

BEST COPY AVAILABLE

Session 9 70
To introduce the concept of the teaching or editing mode.
To introduce students to writing procedures.

Session 10 78
To introduce the editing mode.
To give students practice in editing procedures.

Session 11 83
To introduce the word "debugging" to refer to finding and correcting errors in procedures.
To give students practice in using the editing features in Logo.

Session 12 90
To introduce the concept of one rotation as an amount of turning which is equal to 360°.
To reinforce the concept of procedures as a means of teaching Turtle to follow a set of instruction.

Session 13 97
To introduce saving and recalling of Logo procedures.
To help students develop an understanding of the computer's process of recording information on cassettes or disks.

Session 14 107
To introduce students to problem solving in Logo.
To introduce the primitive for repetition.
To reinforce working with angle measurement.

Session 15 110
To teach students to use procedures (subprocedures) within a procedure.

Session 16 120
To introduce the concept of recursion.

Session 17 128
To allow time on the computer for free exploration and creativity.
To allow time for assimilating and internalizing Logo primitives and geometric concepts at the child's individual developmental level.

Session 18 132
To introduce students to the use of variables as size input for geometric shapes.

Session 19 140
To introduce/reinforce creating circles of various sizes on the computer.

Session 20 147
To help students realize that they have made a tremendous amount of progress.
To make students feel good about themselves.

© Copyright 1985 ICCE

BEST COPY AVAILABLE

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Charts 157
Keyboard Charts 158
Computer Systems Charts 164
Time-On Chart 169
Suggested Use of Time-On Chart 170
Partner Chart 171
Suggested Use of Partner Chart 172
Primitive Commands Chart 173
Suggested Use of Primitive Commands Chart 174
Large Protractor Master 175
Using the Large Protractor Chart 177
Color Charts 178
Key Functions for the Edit Modes 183
PUTTILE Chart 189
Graph Paper Master 190

References 191

Answer Appendix 192

Index 203

© Copyright 1985 ICCE

Session 1

*"The child cannot . . . reconstruct a perspective point of view until he has himself occupied the position to which it corresponds. This is because thought can only replace action on the basis of the data which action itself provides."**

Jean Piaget and Barbel Inhelder
The Child's Conception of Space

OBJECTIVES: To introduce students to the Logo primitives **FORWARD, BACK, RIGHT** and **LEFT** through self-involvement in spatial relationship activities and to reinforce following and giving precise directions.

CONCEPTS/SKILLS: An understanding of commands as directed movement in space using self as a point of reference and conceptualization of commands for future identification with the turtle.

FACILITIES/MATERIALS: Overhead, chalkboard or tagboard strips labeled FORWARD, BACK, RIGHT, LEFT; playground (gym, hallway, etc.); 5x8 card, envelope and sheet of paper for each group of 4-5 students; student copies of Problem Sheet.

PREPARATION: On 5x8 cards, write directions for finding envelopes in various places on the playground, (gym, hallway, etc.). Number corresponding cards and envelopes. For each envelope, prepare a sheet of paper with an assignment to write directions to another specified location. (Sample 5x8 card and Assignment Sheet follow the Activity Section.)

PRIMITIVE CHART:

	APPLE	ATARI	MIT	TI	TRS
Logo Version	Logo Computer Systems, Inc. (L.C.S.I.)	Logo Computer Systems, Inc. (L.C.S.I.)	Terrapin & Krell	Texas Instruments	Tandy Radio Shack
Computer	Apple II (64K) Apple IIe	Atari 400 Atari 800	Apple II (64K) Apple IIe Commodore 64	TI 99/4A	TRS-80 Color Computer TRS-80 Color Computer 2
Primitives	FORWARD BACK RIGHT LEFT	FORWARD BACK RIGHT LEFT	FORWARD BACK RIGHT LEFT	FORWARD BACK RIGHT LEFT	FORWARD BACK RIGHT LEFT

COMMENTS:

One of the central ideas in using Logo to explore geometry is that the turtle uses itself as a point of reference as opposed to using a Cartesian coordinate system. Students' actual experience in walking the commands will help them later in identifying with turtle movements on the screen, especially as the graphic displays become complex. The younger the student, the more essential the self-involvement in spatial activities. Some variations of this activity are listed at the end of this session—however, since debugging a procedure is often facilitated by knowing how to "play turtle," some walking of the commands is recommended.

At this point students have not been introduced to the turtle, so they do not know that they are learning to "play turtle." What they are learning is that four commands can be used to move themselves anywhere on the ground. In addition they are practicing giving and following precise directions in preparation for writing and debugging procedures.

In helping your students learn Logo, there are two important instructional ideas to keep in mind. First, new ideas need to be related to and closely tied in with ideas the students already know. Second, the

*Reprinted by permission of Humanities Press Inc., Atlantic Highlands, NJ 07716.

© Copyright 1985 ICCE

number of new ideas to be presented in one session must be quite limited.

This first session illustrates these two ideas. The words FORWARD, BACK, RIGHT and LEFT are familiar to students. The session consolidates this knowledge, getting students used to the idea of following and giving directions made up from these words.

In Logo, FORWARD and BACK are always accompanied by a number telling how many turtle steps the turtle is to move forward or back. Thus, FORWARD 25 and BACK 18 are correct commands in Logo. Students have knowledge of linear measure and can readily deal with commands such as "take three steps forward," or "take two steps back." This makes it convenient to introduce FORWARD and BACK in a manner both easy for students to understand and consistent with how Logo functions.

The commands RIGHT and LEFT in Logo must be accompanied by a number telling how many degrees to turn. Thus, RIGHT 45 means turn 45° to the right, while LEFT 90 means turn 90° to the left. Many students are not familiar with angular measurement. Thus, this first session introduces the words RIGHT and LEFT but without a specification of how many degrees. The word RIGHT is to mean a full 90° (right angle) right turn while LEFT is to mean a full 90° left turn. Students usually turn 90° when asked to turn RIGHT or LEFT. Make sure they use a 90° turn when following RIGHT or LEFT commands, but do not call attention to it unnecessarily at this time. The difficulties of angular measurement are left to a later session.

The commands RIGHT and LEFT only change direction in place; there is no movement forward or backward. It may be necessary to call attention to this.

PRELIMINARY ACTIVITY:

Print the words FORWARD, BACK, RIGHT, LEFT on the overhead or chalkboard, or use labeled tagboard strips. Discuss the meanings with students.

ACTIVITY:

Part 1

Ask the students to stand, move FORWARD 2 steps, BACK 1 step, turn RIGHT (emphasize that there is no forward or backward movement), FORWARD 1 step, BACK 2 steps, turn LEFT, then FORWARD 2 steps, etc.

Explain to students that they can move from one place to any other by using only these four commands. Ask one student to stand. Give commands to direct that student about the room to various locations. Then let a student give the commands for moving another student to a specific location.

Students also need to see the written commands. Write on chalkboard, calling attention to the single line, vertical listing format; use of capital letters; and the END command:

```
FORWARD 6
RIGHT
FORWARD 3
RIGHT
BACK 1
RIGHT
FORWARD 3
LEFT
FORWARD 6
END
```

Have a student walk the directions. When students demonstrate an understanding of how the directions are written and followed, go on to Part 2.

Part 2

Have each student bring a pencil, and then form them into groups of four or five. Number the groups. Tell them that each group will be given a 5x8 card with directions to a different location on the playground (or gym, hallway, etc.). If they follow the directions correctly, they will find there an envelope with their group number on it. Inside the envelope they will find an assignment asking them to write a set of directions to another specified location.

Pass out each group's 5x8 card. Make students aware that the directions on the card are given using only FORWARD, BACK, RIGHT, LEFT.

Since each person's steps are different in length, tell the students just to walk normally, and they will come fairly close to their envelope. When they find it, they should open it, read it and complete the assignment. Suggest that when they write the commands FORWARD, BACK, LEFT or RIGHT, they should use a new line for each command, similar to the directions given on the 5x8 card.

Variations for Part 2:

- Use a hundreds chart and a marker that can point in a certain direction. The marker might be a small picture of a turtle, with FORWARD being the direction the turtle's head is pointing.

  Place the marker on a square, facing the top of the board. Students are to write sets of directions to move the marker to other places on the board and/or through sequences of places on the board. For example, write directions to move the marker to numbers that add up to 387; move the marker to every number containing a 5; move the marker

© Copyright 1985 ICCE

to every number divisible by 3. Students should practice following the directions written by other students.

- Secure a supply of city maps. Have students begin at a specified point and follow directions of FORWARD 3 BLOCKS, RIGHT, FORWARD 6 BLOCKS, LEFT, etc. Students then identify the intersection by street names. They, in turn, can write similar directions for returning to the original position. (This activity would correlate well with a social studies and map skills unit.)

SAMPLE 5x8 CARD:

Group 1:
Start at tetherball pole facing the school.

FORWARD 20
LEFT
FORWARD 15
RIGHT
BACK 8
RIGHT
FORWARD 20
LEFT
FORWARD 8
END

Remember, each 5x8 card should include the group number and directions to a different location in the same general area.

SAMPLE ASSIGNMENT SHEET:

Complete the directions below using only the four commands FORWARD, BACK, RIGHT and LEFT.

Start at the point where you found your group's envelope, and go to the bike rack.

Begin by facing north. Tell how many steps you take when moving FORWARD or BACK. Remember to write only one command on each line.

Write END when your directions are complete. Sign your names at the bottom of this sheet.

DIRECTIONS TO BIKE RACK
FORWARD_____

ASSIGNMENT: Have students complete the Problem Sheet for this session.

© Copyright 1985 ICCE

Session 1 **Name** _____

Problem Sheet **Date** _____

"Will Hungry Mouse Reach the Food?"

Below is a 9 cm x 12 cm room. Just inside the door and immediately to the right is a wall extending 2 cm into the room. Behind this wall is a dish of food shown by the circled X. You are a HUNGRY MOUSE. You enter the room at the door. You can smell the food but have a problem. **You can only move FORWARD and can only turn LEFT.**

Draw your path to the food. Use a ruler to make straight lines and the end of your ruler to make LEFT turns.

If you succeed in reaching the food, try sending a message to your friend (?) HUNGRY CAT, who also has the problem of only going FORWARD and making LEFT turns. If your directions are correct, HUNGRY CAT can finish the food. If your directions are *not* correct, HUNGRY CAT will not find the food—but might find *you!*

Dear Hungry Cat,

FORWARD _____ cm
LEFT

_____

_____

_____

_____

_____

_____

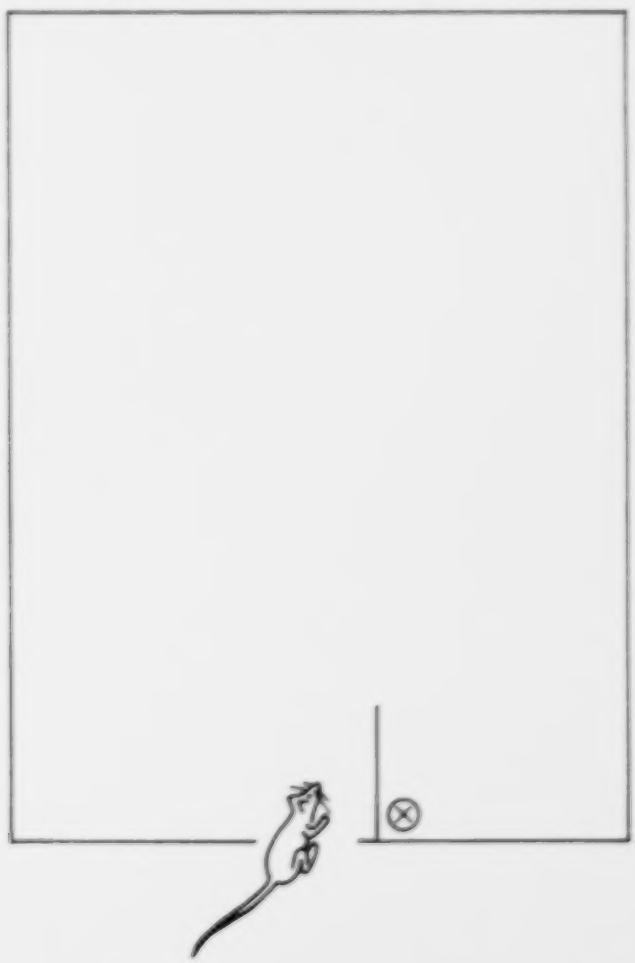
_____

_____

_____

END _____

Your friend,
Well-Fed Mouse



© ICCE 1984. Reproduction for classroom use permitted.

Logo Today: Vision and Reality

by
Uri Leron

Acknowledgement: It is a pleasure for me to acknowledge my profitable and pleasurable collaboration with Pearla Nesher and Rina Zazkis, whence many of these ideas evolved. I also thank the many colleagues and friends who commented on earlier drafts of this paper. Traces of their comments can be found throughout.

INTRODUCTION

Problems

What follows is an informal, personal summary of four years' experience of working, and observing other people's work, with children and computers in a Logo environment.[1] While I remain enthusiastic about the Logo vision of children interacting with the computer and about the potential of this interaction to foster deep and meaningful learning, I am now also aware of some fundamental difficulties on the child's road to acquiring the "powerful ideas" envisioned by Papert.

The current literature contains excellent books on Logo's educational *vision* (Papert, 1980) and *practice* (Watt, 1983), and on aspects of the computer *language* (Abelson, 1981), with most subsequent writings elaborating on the same themes (e.g., *BYTE*, 1982). Yet, little has been explicitly written on *difficulties* in the method, and even less on attempts to overcome them.[2,3] If we are to realize the great potential of Logo, it is urgent that the Logo community turn its full attention to studying these difficulties and the educational problems they pose.

Bugs

In discussing problems in Logo learning, I do not wish to join in an ideological battle over how Logo compares with other languages or other approaches to learning. In fact, as a self-confessed "Logophile" who devotes much professional time to studying and teaching various aspects of Logo, I can hardly claim the status of an objective observer. Instead, I would like readers to consider this paper in the context of one of Logo's central themes on learning programming—a theme here turned toward the study of Logo itself: ***A complex project cannot be expected to run perfectly the first time around, nor should it be abandoned because it has been found to contain bugs.*** On the contrary, by attempting to uncover and understand the bugs, we gain a deeper, more refined understanding of the topic under study. It is natural, then, that we should examine the Logo philosophy for bugs, now that we have "run" it long enough. It is my belief that careful study of these bugs should yield ingenious educational solutions for many of these problems.

Scope

The remarks presented here are based mainly on observations of children in the sixth grade (11-12 years old); their validity, therefore, might be limited roughly to these ages. However, since some of the children were quite bright, it is reasonable to expect some of these phenomena to occur with older people as well, including perhaps mathematically and computationally naive adults. (My experience in teaching Logo to teachers and college students indicates this hypothesis may be at least partially true.) On the other hand, some of the difficulties reported here (especially those related to structured programming) are apparently developmentally related, since they have been much more easily overcome by adults, even those who were noticeably less bright than some of our sixth graders.

THE FUNDAMENTAL TENSION

Unpacking

The various aspects of Logo are presented in Papert's writings as a package deal, and many people—supporters and opponents alike—often treat it as such. For our discussion, however, it will be useful to untie the package and consider these components separately. I refer particularly to two major themes that run through *Mindstorms* and, in my experience, appear to create a fundamental tension in the practice of Logo. These are, in Papert's terms, "Piagetian learning" and "powerful ideas." They address, respectively, the mode and content of learning in Logo.

Piagetian Learning

According to Papert, Piagetian learning is "learning without being taught" (Papert, 1980, p. 7) or "learning without a curriculum" (ibid, p. 31). Children instead build their own intellectual structures through interaction with the environment (the Logo environment, in this case). It is "a learning process that has many features schools should envy: It is effective (all the children get there), it is inexpensive (it seems to require neither teacher nor curriculum development), and it is humane (the children seem to do it in a carefree spirit without explicit external rewards and punishments)" (ibid, p. 42). Papert cites as major examples of Piagetian learning the preschool acquisition of natural language and of early mathematical concepts. One example where we can actually compare formal vs. Piagetian learning is in the acquisition of a foreign language, say French—compare the learning of French in the classroom with learning it by actually living in France.

Powerful Ideas

Powerful ideas are ideas that connect deep and important concepts from mathematics, computer science, problem solving, etc., with what is important to children by giving them the power to achieve goals that they see as impor-

[1] By "Logo environment" I mean the computer language, as well as the educational philosophy and practice as described by Papert (1980).

[2] The Brookline Project Report (Papert et al., 1979) touches on these themes, especially in the detailed reports of the children's work. Also, a number of evaluation studies have appeared recently (see "Other Issues—Logo Research" in this article).

[3] *The MIT Logo '84 Pre-proceedings* (Sorkin, 1984) is a recent notable exception.

BEST COPY AVAILABLE

tant—goals that would be hard or impossible to achieve without the help of these ideas. Examples of powerful ideas are "play turtle" to help in writing and debugging procedures; the use of subprocedures in structuring a complex procedure; and the use of variables to control the size, shape or color of a drawing. The presence of "learnable" powerful ideas in a Logo environment is one of the attributes that makes possible the claim that in Logo one does more than just learn programming.*

Tension

The main thesis here is that the ideal of simultaneously attaining these two goals creates a fundamental tension in the practice of Logo. On the one hand, if we adhere to the ideal of totally spontaneous, non-directed, Piagetian learning, we may find children not acquiring the powerful ideas as much as we had hoped. On the other hand, if we push the powerful ideas too brusquely, we are back to classroom-type teaching with all the notorious side effects (notably, alienation and a feeling of failure).

Hacking

In practice, many Logo teachers interpret Papert's writings as discouraging any direct intervention (such as prescribing certain tasks chosen by the teacher, or other curriculum-type activities) except in response to the child's initiatives. Unaware of the conflict mentioned above, they choose to emphasize Piagetian learning, hoping that the acquisition of the powerful ideas will somehow "happen" to their students. However, my observations indicate that under such conditions most children tend to fall into a "hacking" style of programming which does not seem conducive to learning deep and sophisticated ideas.† This style is characterized by a lot of trial-and-error activity accompanied by little planning or reflection. Yet, it often enables children working on turtle graphics projects to achieve their goals, or change them mid-project, without fully understanding the nature of the bugs they have encountered and how they came to be eventually eliminated.

Now, this hacking style of programming should not *automatically* be condemned. It may be justified in cases where learning difficulties are present, and even for more able children, it may be a natural intermediate phase of exploring the microworld of turtle graphics procedures before going on to a more advanced and systematic programming style. However, it becomes a cause for concern when it persists even in relatively bright twelve-year-olds and bright, sensitive and responsive teachers who have been working in Logo for several months. Surely, under these conditions, children could get more out of their Logo learning.

Curiosity

Learning powerful ideas requires the development of sophisticated new structures in the child's mind, with the necessary concurrent elevation in the level of understanding and reflection. Where

"From an educational point of view, a good study guide seems somewhat paradoxically to enhance students' autonomy rather than suppress it. Properly used, it becomes part of the environment with which they interact, giving them more power to achieve their goals without help from the teacher."

should the *power* which drives this cognitive growth come from? Papert's vision of a Piagetian acquisition of powerful ideas rests upon a model of the child as possessing a burning and powerful curiosity and a need to understand the nature of the phenomena and the bugs appearing on the computer screen. While this may be true of some children (notably those classified as "gifted"), it does not conform with my experience regarding most other children. They are, for the most part, quite content to go on with their programming as soon as they have succeeded in eliminating the bug (often without really understanding its cause). Suggestions to plan ahead or look back are often met with resistance. Similarly, they are often quite content to copy procedures whose products they like (such as circles or fancy REPEAT patterns) from a friend with no apparent curiosity about how the procedure

*That powerful ideas are at the heart of Logo can also be inferred from the inclusion of the term in the subtitle of *Mindstorms: Children, Computers and Powerful Ideas*.

†This non-standard use of the term "hacking" is adopted here for use in the context of novice programmers. It emphasizes an element of indiscriminate, semi-random striking on the keyboard, without pausing to plan ahead or to look back.

works. (Optimistically, one is tempted to speculate on the possibility that children *are* innately curious, and that the above behavior is only picked up during school.)

Educational Challenge

If this description is true, then we must face the major educational challenge of finding more active ways of helping children *without sacrificing the spirit of meaningful and exploratory learning*. Although at this stage we may not know how to actually fix the bugs, based on some very detailed experiments and observations done at the Lab for Computers in Education of Haifa University (jointly with Nesher and Zazkis), I can outline a tentative course of action. It is based on a modified version of Papert's Piagetian learning—I call it "quasi-Piagetian learning" (QPL)—which assumes a more active role for teachers and learning materials, but still leaves plenty of freedom and control in the child's hands. Most important, the exploratory and non-judgmental spirit of Piagetian learning is maintained.

Study Guide

The main external manifestation of this approach has been the introduction of a carefully written study guide. Far from being just a manual for the technical aspects of the language, the guide helps children learn powerful ideas by partially directing their activities and then prompting them to reflect on their results. To illustrate, in order to learn the effective use of subprocedures as an introduction to structured programming, we presented the children with drawing projects composed of the same components with varying interfaces, and then discussed with them the question, "What is similar and what is different in these drawings?" Another approach asked the children to "teach the turtle" a relatively complicated drawing (a person, for example), and then asked them to modify certain parts in certain ways (e.g., redraw the legs, hands and head in different colors). Comparison of the various procedures written by the children enabled us to convincingly illustrate the merits of modular programming.

From an educational point of view, a good study guide seems somewhat paradoxically to *enhance* students' autonomy rather than suppress it. Properly used, it becomes part of the environment with which they interact, giving them more power to achieve their goals without help from the teacher.

Quasi-Piagetian Learning (QPL)

The transition from pure to quasi-Piagetian learning represents a deliberate action to trade off some of the freedom inherent in Piagetian learning for a deeper understanding of the ideas behind the programming activity. (Again, our experience has shown that *some* compromise must be made, as it is unlikely that we should fully achieve both.) Still, we were able to leave plenty of freedom and control with the children. First, they worked on a "free" personal project about one third of the time at the computer, along with work from the study guide. Second, even when working on the study guide, the children were left to evaluate their own projects. Thus, our answer to "Is this house OK?" was "Do *you* like it?" (In other words, "If it's good enough for you, it's good enough for us.") Needless to say, there were no tests or grades. Significantly, the children did not seem to mind the more structured environment, and in fact some seemed to prefer working on the study guide projects to working on their own. Again, somewhat paradoxically, one might say that QPL recognizes the *strengths* of real-world children, as well as their weaknesses: They can take and even *like* some teaching if it is done in the spirit indicated above.

> ***"... it is possible that the more prevalent problem with Logo will not be too little learning ... but rather too much teaching—and of the wrong sort."***

It should be noted that the meaning of QPL is not fixed or final, and is likely to change as our understanding of the teaching/learning process improves. It is also hoped that individual Logo teachers will create their own definitions of QPL, reflecting their personal styles. The above definition of QPL merely represents *my* current view of where that elusive entity, the golden path between a subject-matter-centered approach and a learner-centered approach, might lie.

The Teacher's Perspective

The introduction of the study guide had two added benefits. It proved very helpful in approaching the ideal of maximum autonomy for children—that is, independence from teacher help. As a rule we refused, good-naturedly but firmly, to answer questions whose answers could easily be found in the book. Thus, to the usual list of heuristics there was added the useful reminder, "Have you looked in the book?"

The guide was also a substantial help to teachers. Teaching Logo properly requires a highly skilled teacher, and is seldom achieved without extended training by a Logo expert—something that is hard to effect on a large scale. Because of this, many Logo classes are being taught by poorly prepared teachers. Such teaching tends to concentrate on technical and syntactical aspects, neglecting broader aspects of the language and the educational philosophy. Moreover, even well-trained Logo teachers find it difficult to implement the Logo philosophy when working with a *group* of students.

In view of these problems, it is desirable to delegate to the study guide as much of the task of assisting the children's learning as possible. A book cannot, of course, replace a good teacher. But a good book can *help* all teachers, and it can make a great difference in the case of a poorly prepared teacher. Again, the teacher can now concentrate on helping students reflect on and verbalize (and thus, conceptualize) their computer experiences, rather than actually carry them through these experiences. When working with a group, reflection can be encouraged through a group discussion following a common activity.

Logo Abuse

A word of caution should be added here. This article is addressed mainly to the "Logo community," where the Logo philosophy is more or less taken for granted or is, at least, well known. Hence the emphasis on debugging Logo to better fit reality. However, debugging does not mean rejecting, and one should be just as wary of the reverse phenomenon. As the popularity of Logo increases, there is a very real danger that its popularity as a "convenient computer language for young kids" will outrun its popularity as a philosophy of education and a carrier of powerful ideas. Indeed, it is easy for anyone with enough motivation, a microcomputer and a Logo manual to learn the technicalities of the language or, for that matter, to teach it. Learning the powerful ideas embedded in Logo, and even more so the educational philosophy behind it, is much harder and will not generally happen without some expert guidance. Thus, it is possible that the

BEST COPY AVAILABLE

more prevalent problem in the practice of Logo will not be *too little learning* (as addressed in this article), but rather *too much teaching*—and of the wrong sort.

So, while I suggest that we try—cautiously and tentatively—to add some teaching and curricular activities in order to increase children's learning of powerful ideas in Logo, it should be made clear that "teaching" does not mean lecturing, "directing children's activities" does not mean forcing, and "curricular activities" does not mean a fixed and rigid curriculum. In other words, while experimenting with quasi-Piagetian learning styles, let us not fall into an anti-Piagetian one.

SUBPROCEDURES AND MODULARITY

Most children pick up easily and naturally the first steps of Logo programming—navigating the turtle and writing simple procedures (that is, entering the editor, writing text, exiting the editor and running the procedure). The next major step, however—the use of subprocedures, modularity, and the whole cluster of intellectual tools commonly known as structured programming—already poses serious problems. This topic will now be used to discuss more specifically the general learning issues raised previously.

There are two reasons why the topic of subprocedures and modularity was selected for a detailed discussion. First, it includes some of the deepest and most powerful ideas that children can learn in Logo (and incidentally, hardly anywhere else); and second, due to the Haifa University study with Nesher and Zazkis mentioned earlier, we simply have more information on this than on other topics such as variables or recursion.

Levels

When talking about learning to use subprocedures, or any of the other powerful ideas in programming, at least three levels of such learning should be distinguished. The first and most superficial level is the *syntactical* level. Here the child simply learns a set of syntactical rules (for instance, the rule that user-defined procedures can legally "call" each other). The second or *semantic* level enables the child to actually *use* these tools; it requires the cognitive ability to conceive of a complex procedure as a hierarchy of subprocedures with interfaces between them, as well as the technical ability to program in this style. The third level, which might be called the *expert* level, is the hardest of all to achieve. Here, in addition to the requirements of the semantic level, children also show the tendency to use these tools spontaneously and willingly on their own. Hopefully they have now begun to appreciate the power and perhaps also the beauty of the structured programming method.

Children's Learning

Here is a tentative sketch of our observations of what children actually learn in relation to the above three levels.

The syntactical level seems to pose no special learning difficulties: Most children will occasionally insert one procedure inside another, indicating that they know such use is "legal." The semantic level appears to be quite sophisticated—and harder to achieve—than previously suspected, and it is not usually attained through Piagetian learning alone —at least not within the age and time limits discussed here. The need to distinguish between the semantic and expert levels arose in the Haifa University study, in which several bright sixth graders received a thorough introduction in the quasi-Piagetian style to the ideas and practice of subprocedures and modularity in programming. The children acquired sufficient proficiency with subprocedures and their interfaces (or "seams" as we called them) to structure their procedures *at our request*—often with the air of, "Oh, well, if you insist. . . ." They could even quote at our request the merits of this style of programming. However, it was clear that they did it, at least in part, to make us happy, and they would often return to their linear style when left alone. Thus, we might say that they attained the semantic, but not the expert, level.

"On the one hand, if we adhere to the ideal of totally spontaneous, non-directed, Piagetian learning, we may find children not acquiring the powerful ideas as much as we had hoped. On the other hand, if we push the powerful ideas too brusquely, we are back to classroom-type teaching with all the notorious side effects...."

Note: Critics of the quasi-Piagetian approach (from the Piagetian side) might claim that these results are to be expected: Had we left the children to "grow" naturally into structured programming rather than "pushing" them into it, they would have used it more willingly and naturally. This claim rings true, but the reader should be reminded that quasi-Piagetian learning was introduced precisely because in reality we did not see students growing even into the semantic level of learning these concepts, let alone the expert level.

On the other hand, I in no way wish to imply that the topic is intellectually too hard for these children. The message, rather, is that the topic is more sophisticated than we suspected, and that we'll have to think harder on how to help children acquire such skills. This last point also applies to the other problems discussed in this paper.

Resistance

What are the sources of the resistance our children have shown to structured programming? Here are a few speculations.

For one thing, their projects were not complex enough to really make clear the need for structuring. (For us, this style has acquired a strong *aesthetic* appeal, so we use it even in the simplest of programs. The children, however, do not yet see the beauty of it, so they won't use it unless the need becomes apparent.) Furthermore, we suspect that at this stage of their learning, children may see a structured programming style as harder to use than their more natural linear style. One reason is their lack of a clear concept of the *interface* between any two subprocedures, and the related importance of the turtle's *state* before and after each subprocedure. This difficulty, commonly observed in our work with children as a major source of bugs, could also have been expected on theoretical grounds: When conceiving of a given picture as a hierarchy of sub-pictures and interfaces, the sub-pictures can be directly perceived in the original picture, whereas the interfaces cannot. The interface, representing a *relationship* between sub-pictures, lies in a more abstract realm than the sub-pictures themselves, and hence is harder for students to grasp. Finally, there is the related difficulty, experienced by our young novice programmers, of debugging structured procedures. This somewhat surprising observation will be discussed in the next section.

Note: In the Brookline Project Report (Papert et al., 1979) there seems to be no clear distinction between these levels of learning to use subprocedures. One of the deeper levels is probably meant when this skill is declared to be one of the conditions for saying that the students "have 'learned to program' in any significant sense" (p. 1.14), while the more superficial level is probably meant in the following quote from the Summary of Findings (ibid): "Which students learned to program? [. . .] all students except those in the lowest quartile of school performance did reach our criteria." (My conclusion is drawn from the detailed reports of the students' work. The relevant criterion is [A3] on p. 1.5.)

DEBUGGING

Importance

The issue of debugging and the educational problems it poses are of the utmost importance; here our attempt to encourage the development of independent, critical thinkers is put to its hardest test.

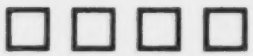
We first note that the real test for understanding a programming concept is not merely the ability to write successful procedures involving that concept, but rather the ability to debug such procedures. For example, it is easy to write certain successful procedures involving recursion with only a vague understanding of recursion (many novice programmers do that). It is only in *debugging* recursive procedures that the full intricacy of the concept is revealed. For example, even experienced programmers will sometimes get lost in debugging an innocent-looking pair of procedures that recursively call each other. Also, debugging appears to be closely related to pure problem solving, so it is not likely that we can find easy ways to teach all children to be proficient debuggers (and thus, programmers). Here are a few common debugging-related difficulties we have observed.

Modularity and Debugging

It is well known that long, unstructured procedures are hard to understand and debug. Somewhat unexpectedly, many of our novice sixth-grade programmers seem to have found modularized, structured programming just as hard. This is important, as it makes the educational task of "converting" them to the latter style of programming harder for us. One possible explanation is that the well-known advantages of structured programming depend on a fuller understanding of the process than our children usually possess. At this early stage, simple step-by-step programs may be easier to construct and debug by trial and error, directed by a vague, partial understanding of the program and bug.

"The Seam"

A second debugging-related problem has been mentioned before: It is the lack of a clear concept of the interface (or the "seam," as we have called it in our work with children) between subprocedures, and the role of the turtle-state in determining this interface. (Specifically, the interface between any two subprocedures consists of the steps necessary to move the turtle from its state at the end of the first subprocedure to its state at the beginning of the second.) For example, consider the problem of teaching the turtle to draw the following picture:



It would be nice if children eventually wrote something like the following:

```
TO SQUARES
  MOVE.TO.START
  REPEAT 4 [ SQUARE MOVE.NEXT ]
END

TO MOVE.TO.START
  HT PU
  LT 90 FD 100 RT 90
  PD ST
END

TO SQUARE
  REPEAT 4 [ FORWARD 40 RT 90 ]
END

TO MOVE.NEXT        (This is the seam.)
  PU
  RT 90 FD 70 LT 90
  PD
END
```

Many children find this task very hard to do, and when they eventually get it right after much trial and error, it usually does not look anything like the above program. Not only do they not think of the task in a top-down fashion, but even when we supply them with the top level (the procedure SQUARES above), they still have considerable difficulty in completing the interface MOVE.NEXT.

Incidentally, I do not attach special educational value to the top-down style of programming. I do think, though, that it is important for us and for students to be aware of the two polar programming modes (top-down and bottom-up) and of the possibility of choosing an appropriate mixture of them to suit the given situation and one's personal style.

Tracing

The third difficulty is in "playing turtle"—that is, formally and rigorously tracing the steps of the procedure as the computer would. There is evidence that this activity is hard for children, probably due to interference from their intuitions and intentions. It might at first appear strange that children who can *write* programs will fail to *read* and *interpret* correctly programs of the same difficulty level. However, a related phenomenon has been reported in the case of producing *natural* language vs. understanding it. (Donaldson, 1978, pp. 72-75, especially the report by Lois Bloom of "a child called Peter, aged 32 months, [who] proved unable to imitate successfully a number of sentences he had himself spontaneously produced the day before.") The problem seems to lie not so much in mastering the turtle's language as in understanding the implications of its being a *formal* language.

As before, the suggestion is not that the skills discussed here are too difficult for children to acquire, but that they are not normally learned through Piagetian learning alone. It does seem possible to specifically teach such skills through interesting and carefully thought-out projects—that is, through quasi-Piagetian learning.

MATHEMATICS

Learning powerful mathematical ideas in Logo is even more involved than learning computational ideas, and I shall treat it here only briefly.

While an expert watching students work in Logo can clearly see a lot of mathematics going on, it is not clear how much of it is actually being learned. That is to say, the turtle may be a math-speaking creature, but we cannot automatically assume that students are always listening to what it is saying. As with other powerful ideas, interaction within a more structured environment may be needed if the children are to *acquire* these mathematical concepts. In other words, while maintaining Papert's view of children's learning through interaction within a microworld, it may be helpful to think of, and experiment with, more tightly structured microworlds than, say, that of turtle geometry.

Reporting

At the very least, it is important when reporting on students' learning of

mathematics in Logo that a clear distinction be made between the mathematics observed by us and the mathematics learned by the child. To give an extreme example, while there is a lot of group theory going on in students' Logo work (Leron, 1982), one could not sensibly claim that students are actually learning group theory. In the Logo literature, this sort of distinction has not always been made clear.

The mathematics we see in the Logo environment is mathematics *potentially learnable* by children; whether they actually learn it is again a major educational challenge for Logo. The difference is often one of levels: The learning process should elevate the child from merely being engaged in mathematically rich activities to actually forming mathematical concepts.

Coping

I see two ways we can address this problem. The first is devising more tightly structured microworlds, each perhaps embodying one set of related powerful ideas such as polygons and elementary number theory, or functions and variables. Hopefully, such microworlds can be constructed that will be sufficiently interesting from the point of view of both the mathematics and the child.

Second, it is important that programming work be followed by verbal or symbolic work in order to elevate the knowledge acquired by "doing" to a more conceptual level. I believe with Papert that "children learn best by doing and by thinking on what they do" (Taylor, 1980, p. 161). I also believe that the advent of inexpensive, powerful computers is bound to solve the "doing" part of this learning process. However, the "reflecting" part still poses serious educational problems and has been somewhat neglected in Logo work.

In practice, it is hard to maintain the sort of Piagetian interaction depicted in *Mindstorms*, if only because teachers work mostly with a group of children, rather than a single child. When working with a group, a class discussion following a common activity is a good way to "reflect." Other Logo educators have experimented with social interaction between the students—for example, between pairs working together at the computer—as a means of verbalizing and reflecting. This seems promising, provided the interaction is a constructive and cooperative one. (See Hoyles and Sutherland, "Intervention Strategies and Collaboration in Learning Logo," in Sorkin, 1984.)

Of course, a class discussion, or even working in pairs, has its own problems: Some students become passive or go off daydreaming and are likely to fall behind. This is no surprise—the acquisition of deep and sophisticated ideas is a complex process in which schools in general have failed by and large. If in Logo we succeed with more students and make the learning experience more meaningful and positive, we have achieved a lot. We should not expect miracles, nor should we consider anything less than a miracle a disaster.

OTHER ISSUES

The topics raised in this section deserve a fuller, more detailed treatment. Some of the issues will be mentioned here, with no attempt to be comprehensive.

Variables, Recursion

Full understanding of these topics requires a degree of formal thinking that is not easy for most sixth graders; good, concrete metaphors such as "mail boxes" for variables or "little people" for recursion are very helpful (Watt, 1983, Chapter 7). Even then, we should not expect full understanding of these topics within the first few months, or even a year, of the children's Logo work. This is natural and should not be cause for concern; in fact, a good way to learn a sophisticated topic is to pass through several stages of partially formed knowledge, gradually refining understanding. This does, however, raise one major educational problem: Since incomplete understanding makes debugging difficult or impossible, this situation is likely to make the child dependent on the teacher or other expert for debugging. So while we allow for stages of incomplete knowledge, we must at the same time find ways to preserve children's independence. It would be best, of course, if we could *use* these very debugging difficulties to further the child's understanding. Unfortunately, this is not always possible, for while the deeper understanding required for debugging may take months to achieve, the debugging problems themselves must find an immediate solution if students are to continue their projects.

"... we must face the major educational challenge of finding more active ways of helping children without sacrificing the spirit of meaningful and exploratory learning."

In introducing these topics, variables appear most naturally as inputs to procedures (in particular, as local variables), where they control easily observable graphic properties. The direct assignment command MAKE can and probably should be postponed until later. Except for initial minor syntactical difficulties, we did not observe any serious problems at this stage. However, some of the issues concerning outputs, scoping and the passing of variables between procedures are complex and possibly too difficult for most beginners.

Recursion should be approached with particular care, as it conceals very sophisticated and complex structures beneath innocent appearances. Many students I have taught or observed tend in such cases to learn by rote certain "idioms" and the effects they produce, without really understanding how they work. While this may be a natural way to enter a new topic, it makes good planning and debugging almost impossible, so it may inhibit progress if it persists for too long. Recently there have been several attempts to address the question of learning recursion in Logo (e.g., *The Computing Teacher,* 1983-84). While these sources offer good ways to *explain* recursion, and are thus useful for teachers and bright students who can figure out complex issues for themselves, they do not go far enough in examining the difficulties most learners have with recursion. What we need most is a better understanding of the psychological issues involved in thinking recursively, and the educational issues of enhancing such thinking in novices.

Beyond Turtle Graphics

The non-graphic parts of the Logo language, and especially projects based on list processing, are, in my experience, rather hard for most sixth graders. It is relatively easy to introduce lists and the elementary commands for taking them apart and putting them together, but most interesting projects require a sophisticated combination of list-processing commands, output and recursion —a rather explosive combination at this level. (Music processing is an interesting and perhaps somewhat easier gateway to lists.) Fortunately, graphics programming in Logo, and the ideas that can be learned with it, are rich enough to easily

and profitably occupy the students, with appropriate guidance, for a whole year, by which time they may be ready to move on.

Logo Research

There has always been a strong demand, especially from the outside, for the Logo community to supply research data to prove its grand claims regarding the educational value of Logo. Several "evaluation studies" have recently appeared, mostly with negative findings. Such evaluation research ("Is Logo successful in teaching kids problem-solving skills?") is, in my view, irrelevant, or at least premature at this stage. First of all, Logo is an educational *tool* of great potential, but its success (and any evaluation of it) depends primarily on *how* we use this tool. There is no sense in which one can "evaluate" Logo. At best, having specified a well-defined *method* of using certain parts of Logo, one can try to evaluate that method. Yet, most research on Logo fails to specify the precise method used. Second, the duration of studies done to date has usually been too short to reasonably expect measurable results.

Third, the evaluation tools used are, for the most part, inappropriate for the processes which are important in Logo learning; thus, they may fail to demonstrate viable learning even when such learning has occurred. For example, most students we have worked with seem to have gained some sort of a vague, partial understanding of many powerful ideas. This kind of learning is very valuable, indeed a *necessary* intermediate step toward a more complete understanding, but it is unlikely to be detected by standard evaluations, where lack of complete understanding is often rated as complete lack of understanding.

So we first need to consider in detail the theoretical and practical problems that accompany the everyday practice of Logo and examine alternative ways of coping with these problems. Only then can we hope for a meaningful evaluation. It is true that claims, grand or otherwise, need substantiating, but in view of the research problems indicated, the Logo community should respond not by supplying "proofs" but by making its claims more moderate. The kind of research very much needed at present is detailed studies and documentation of various issues related to children's learning, and insightful, responsible reporting of findings from these studies. The success of Logo will eventually depend more on whether we can answer honestly and insightfully questions of the type raised here than on formal demonstrations of its achievements.

CONCLUSION

I wish to conclude with an optimistic note. The problems discussed in this article were all observed in cases where teacher intervention was minimal, with very little planning or directing of the child's activity. Just as significantly, the time span of their Logo work was short —usually no more than four months, or about 30 to 40 hours at the computer.

The problems discussed are not insurmountable. Indeed, one envisions a well-planned, modular, spiral course extending over perhaps several years (so that there is plenty of time for children to experiment and grow naturally), accompanied by well-written and imaginative learning materials that children can refer to when needed, thus rendering "teaching by lecturing" almost unnecessary.

The development of such materials is not an easy task if we are to preserve the spirit of exploratory and personally meaningful learning. But in my view it is one of the most urgent tasks confronting the Logo community if we are to make the Logo vision a more realistic one. It is reasonable to expect that such a course will have successfully dealt with the above-mentioned problems, thus greatly increasing the intellectual benefits children derive from their Logo work.

However, it should again be emphasized that the quasi-Piagetian style, though giving a more active role to teachers and learning materials, is nothing like "back to basics." It is still closer to Papert's Piagetian learning than to the sort of teaching normally found in schools.

Finally, the problems discussed here are not unique to Logo learning. They are likely to appear whenever we make a serious attempt to satisfy the demands of both the learner and the subject matter. Thus, Logo serves also as a testing ground for various experimental educational approaches—a lab for learning about learning. One of the great attractions of studying children's Logo learning is that if such a "golden path" exists at all, this seems an ideal place to look for it. **END■**

[Uri Leron, Department of Science Education, Technion—Israel Institute of Technology, Haifa, Israel.]

REFERENCES

Abelson, H., *Logo for the Apple II* (or *Apple Logo*), *BYTE*/McGraw-Hill, 1981.

BYTE (Special Logo Issue), Vol. 7, Number 8, August 1982.

Donaldson, M., *Children's Minds*, Fontana, 1978.

Leron, U., "The Group of the Turtle," *BYTE*, Vol. 7, Number 8, August 1982, pp. 330-331.

Papert, S., D. Watt, A. diSessa and S. Weir, "Final Report of the Brookline Logo Project, Parts II and III," Logo Memos 53 and 54, M.I.T., 1979.

Papert, S., *Mindstorms: Children, Computers and Powerful Ideas*, Basic Books, 1980.

Sorkin, R. J. (ed.), *Logo '84, Pre-proceedings of the 1984 National Logo Conference*, M.I.T., Laboratory for Computer Science, June 1984.

Taylor, R.P., *The Computer in the School: Tutor, Tool, Tutee*, Teachers College Press, 1980.

The Computing Teacher (Special Logo Issue), Vol. 11 No. 5, December-January 1983-84.

Watt, D., *Learning with Logo*, *BYTE*/McGraw-Hill, 1983.

BEST COPY AVAILABLE

Kathleen Martin and Tim Riordon

Welcome back! The summer has provided us with time to rethink and reorganize. Consequently, ***The Logo Center*** *will take on a bit of a different look this year. We've decided that there are at least two ways to think about Logo. You can* ***teach Logo,*** *and you can* ***use Logo to teach other things.*** *One way concentrates on learning the capabilities of the language. The other way uses Logo as a tool to investigate some concept. Both are important and, in actual practice, the distinction is not as simple or as clear as we've defined it. However, we do think it is an important distinction for educators to keep in mind.* ***The Logo Center*** *will attempt to address this double perspective through a two-part approach. The parts may often be unrelated. This is different from last year's thematic approach, but will sometimes enable us to present some of the more advanced dimensions of Logo and, at the same time, take better advantage of some of the excellent ideas which you, our readers, will want to submit.*

There is a real need for procedures which exemplify good use of the list processing capabilities of Logo. We encourage those of you who have written such procedures to send them to us. Since many versions of Logo are already available and others are anticipated, ***please make sure you identify which version you are using.*** *Also,* ***please write your procedures in readable fashion,*** *avoiding abbreviations and including explanatory comments. Although we are certainly interested in how you may have used these procedures with your classes, our space restrictions within the column will not enable us to include such descriptions. We are eager to hear from you and anticipate that some of you will be seeing yourselves in print in future months.*

Researching Spirolaterals

An Historical Prelude

Somewhere in your work with computers, you and your students will probably want to study the history of such a powerful technology. Depending upon the age of your students, you might encourage them to research the characters and situations that contributed significantly to the computer's development. You'll certainly want to include Charles Babbage and his analytic engine. In studying Babbage, you'll encounter Ada Augusta, the only daughter of the poet Byron, whose exceptional mathematical ability enabled her to understand and to program Babbage's machine. A significant event to examine is the 1890 census. Since it had taken seven years to hand-compile data from the 1880 census, Herman Hollerith invented the punch card machine, an electromechanical forerunner to the computer. The 1890 census was completed in six weeks! Don't forget to look for Hollerith's connection with IBM. Alan Turing is another important figure. Although his Turing machine was purely theoretical, its conception laid the mathematical groundwork for future development of the general-purpose computer.

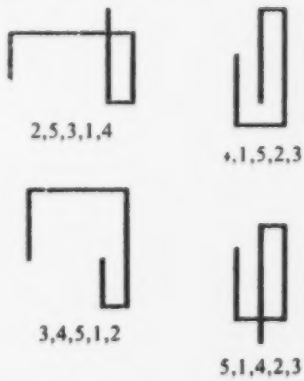
With World War II there emerged serious problems for which the computer seemed the inevitable solution. The British Colossus series was a group of electronic computers used in cryptography (code-making and code-breaking). Americans John Mauchly and J. Presper Eckert designed ENIAC, a machine capable of quickly computing the thousands of mathematical operations needed to compile ballistic tables for guns and missiles. ENIAC had many of the characteristics of modern day computers with the exception of storage capability. The development of the stored program was the contribution of John Von Neumann. Once the Von Neumann concept was wed to the ENIAC technology, the present day computer revolution began in earnest.

As you discuss the historical development of the computer, students quickly become aware of how it has contributed to the natural, social and military sciences. Modern day computers can readily solve the problems that confronted the inventors of their precursors: Babbage's perturbation with errors in logarithmic tables; Hollerith's frustration with the slowness of processing census data; Mauchly and Eckert's consternation with the mammoth number of ballistic calculations.

While students easily grasp these applications, they are not as prone to realize the significant impact of computers on research, particularly mathematical research. There is no better way to awaken them to this fact than to have them *participate* in such research, using the computer as a tool to facilitate their investigations. In simulating the work of mathematicians, students can actually experience the way in which the computer removes tedium and mental drudgery from their work, thereby freeing them to concentrate on important patterns and relationships.

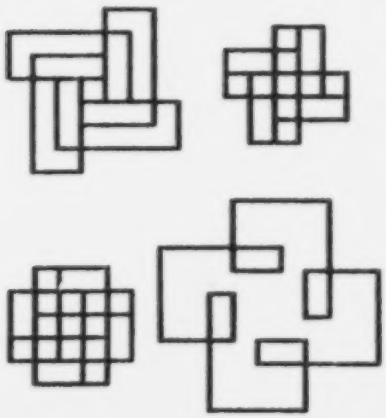
The Search Begins

There are any number of mathematical topics that you and your students might choose to research. The one we've chosen is spirolaterals. Describing a spirolateral is easier done than said. Begin by giving the students graph paper and instructing them to write the numbers 1 through 5 at the top of the paper in any sequence. Tell them each number is the length of a line. Then have them graph the sequence using a 90-degree right turn between each member, i.e., a clockwise turn. Graphs of some possible sequences are shown below.

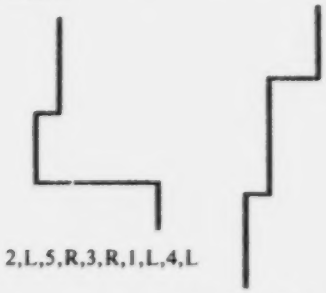


Tell the students to repeat the sequence until they return to their initial position and heading, keeping a record of the number of repetitions. Remind them to always turn before beginning the sequence again. If the students graph

their sequences properly, they will all repeat the sequence four times before returning to their initial state. Their designs, however, may be dramatically different. Designs generated in this way are called spirolaterals. Each number in the sequence is a side.



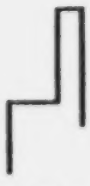
Students usually get excited about the fact that *everyone's* sequence results in a spirolateral after four repetitions. Someone usually wants to know what will happen if the turns are not restricted to right turns. Tell the students to graph their same sequence again, but this time to mix left (counterclockwise) and right (clockwise) 90 degree turns between the numbers.



2,L,5,R,3,R,1,L,4,L

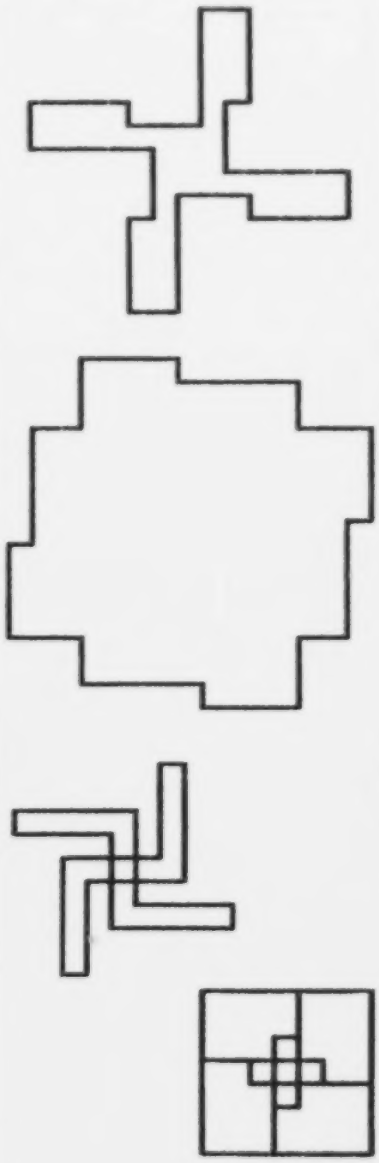
4,R,1,L,5,R,2,L,3,R

3,R,4,R,5,R,1,L,2,R



5,L,1,L,4,R,2,L,3,L

The consequent spirolaterals are even more varied, but once again every student will have repeated the sequence four times before returning to his beginning point.

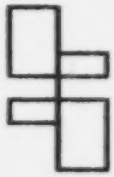


Hopefully, your students will have begun to wonder aloud why all of their sequences close after four repetitions despite the many variations in their designs.

Tell them to add another number to their sequence. Make it a small number like 1 or 2 rather than 6 in order to keep the spirolateral on the paper. They found that the 5-sided spirolateral with 90 degree turns closed after four repetitions of its basic sequence. What happens with a sequence of 6 numbers? (Your students might want to use all right turns again to simplify the graphing.)

5,2,1,4,3,2

The students will discover that every 6-sided spirolateral with 90 degree turns, whatever the sequence of numbers and/or turns, closes after two repetitions of the sequence.



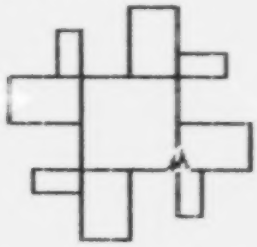
By this time everyone will probably be complaining of the difficulty of keeping track of moves within these more complex sequences. It's tedious work, something that computers do quite well (and without any complaining). Suggest to the students that they write a Logo procedure to draw the sequence for them. Since a turtle step is a very small amount, multiply the numbers in the sequence by 10.

```
TO SPIRO
FD 50 RT 90
FD 20 RT 90
FD 10 RT 90
FD 40 RT 90
FD 30 RT 90
FD 20 RT 90
END
```

The turns need not be all right turns; they could be all left or a mixture of right and left. Why doesn't it matter?

The student types "SPIRO," hits RETURN, and the sequence is drawn. Now students are free to attend to other things, like the number of sequences and the heading of the Turtle. To investigate a sequence of seven numbers, your students need only add a line to their procedure: FD 30 RT 90.

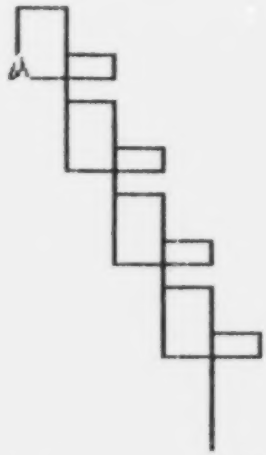
How many repetitions are required before a sequence of seven numbers closes to make a spirolateral?



Why does a 7-sided spirolateral require the same number of repetitions as a 5-sided spirolateral?

Again add a number to the sequence: FD 20 RT 90

What happens when they try to draw an 8-sided spirolateral?

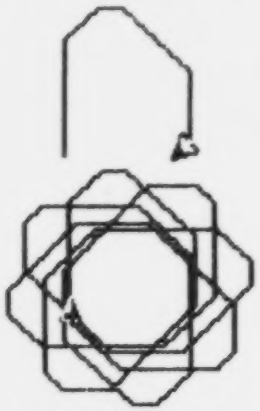


It's impossible to make an 8-sided spirolateral. The sequence never closes. Why not? They might get a hint by comparing the first sequence of the 7-sided and 8-sided spirolaterals. Note that the graph of the sequence looks the same except for the position and heading of the Turtle. Can you predict what would happen with a 4-number sequence?

If the computer has been helpful in examining right-angle spirolaterals, it is invaluable in examining spirolaterals with less convenient angles. Your students can easily modify their Logo procedures to handle any size angle. It's simply a matter of using a variable angle in the procedure.

```
TO SPIRO  ANGLE
FD 50 RT  ANGLE
FD 20 RT  ANGLE
FD 10 RT  ANGLE
FD 40 RT  ANGLE
FD 30 RT  ANGLE
END
```

With the above procedure your students are confined to exploring 5-sided spirolaterals, but with any size angle. What happens with a 45 degree angle?



The sequence requires eight repetitions before it closes. Add another number to the sequence: FD 40 RT ANGLE

Encourage your students to try to predict how many times the sequence will have to be repeated before a spirolateral results. Tell them to concentrate on the heading of the Turtle.

As your students explore spirolaterals, help them to organize their findings in a table so that they can more easily observe the patterns and relationships that emerge. The rows can be the number of sides, the columns can be angles (30, 45, 60, 72, 90, 120, 144, 150), and the entries will be the number of repetitions necessary.

Each time students want to investigate a spirolateral with more or fewer sides, they will need to edit their procedure. This can be avoided by rewriting the procedure to include more variables. Since they have found that the sequence of numbers is not significant in determining the number of repetitions necessary to form a spirolateral, the sequence could begin with a DISTANCE of 10 and simply add 10 for each additional side. Then the number of sides would be the FINAL.DISTANCE divided by 10. The procedure could look something like this:

```
TO SPIRO  DISTANCE  ANGLE
   FINAL DISTANCE
IF  DISTANCE > FINAL DISTANCE
  [ STOP ]
FD  DISTANCE
RT  ANGLE
SPIRO  DISTANCE + 10  ANGLE
   FINAL DISTANCE
END
```

Using this modified procedure, all spirolaterals will look alike (assuming the same inputs). However, the ordered sequence (10, 20, 30, 40, 50, etc.) makes it easier to observe what is happening in terms of the position and heading of the Turtle, as well as eliminating tedious editing.

After completing the table, your students will notice all sorts of interesting things. Why does a particular pattern seem to emerge for a specific angle? Which angles seem to have related patterns? Why is this so?

It's not really necessary for your students to understand all of the "whys" behind spirolaterals. Some answers to some of the questions can become rather mathematically sophisticated. It may be enough that they observe patterns and appreciate the relationships that they see. Mathematical research is an ongoing endeavor, and the computer has become an indispensable tool in assisting in that effort. Spirolaterals have a history of their own before the invention of the computer, as do most mathematical topics. The computer offers spirolaterals, as well as many other areas of mathematics, the possibility of a richer history. Perhaps this activity will encourage your students to become more active participants in such history-making. **END■**

[Kathleen Martin, Dept. of Education, University of Dallas, TX 75061.]

[Tim Riordon, Assoc. Dir. of Curr. and Trng., National Computer Training Institute, 39510 Paseo Padre Parkway, Fremont, CA 94556.]

BEST COPY AVAILABLE

Adventure Stories Revisited

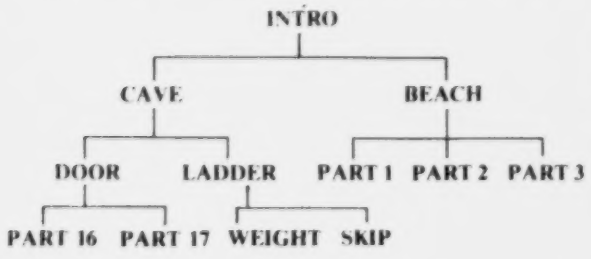
Last year we suggested a project called "choose-your-own-adventure stories." Many readers found this to be an interesting, but frustrating, project: "OUT OF SPACE—PLEASE ERASE SOMETHING!" With the help of Jock McLees of Terrapin, Inc., we have found a solution to this problem. It is possible to create large text files that take up almost no Logo workspace. These files can be manipulated using a few Logo procedures. You can begin applying this technique whenever you have a Logo project which will involve a page or more of instructions. The instruction page can be a text file instead of being a series of print statements in a procedure. The choose-your-own-adventure project will also provide an opportunity for readers to learn more about how variables can be used in Logo programming.

Choose-Your-Own-Adventure Story

Choose-your-own-adventure stories are popular with students of all ages. In a choose-your-own-adventure the reader participates by making a choice. A short chapter—perhaps only a page or so of text—usually ends with a choice like:

> If you decide to go into the tunnel, turn to Ch. 8.
> If you continue down the river, turn to Ch. 11.

Choose-your-own-adventure stories differ from regular books in that they do not proceed from Chapter 1 to Chapter 2 to Chapter 3, etc. You could think of a book of this type as having a tree-like table of contents rather than a linear table of contents. Here is an example which we will often refer to:



Notice that the end of each story section presents an option to the reader. At this point the story branches. The reader makes a choice and, in so doing, enters into the story. Many classrooms have created stories like this as a writing project. It's a good project for a small group of students. The whole group can write the introduction and present a branch option. Half the group can develop one branch while the other half works on the other branch. Each time an option is presented, the group working on it can divide again. Finally, individual students will be writing individual "chapters." Let's consider how to do such a story on the computer.

Describing the Task

When you have just a sketchy idea for a Logo project, a good first step is to state more precisely what the final product will look like.

- The program will let the reader make the choices.
- The program will not run out of memory.
- The program will let the reader show Logo illustrations.
- The program will let the reader quit.
- The program will not let the reader skip ahead. In other words, if the choices are GO INTO THE TUNNEL or CONTINUE UP THE MOUNTAIN, the reader is not permitted to choose CLIMB THE LADDER. Climbing the ladder may be a legitimate choice at some other point in the adventure, however.

The purpose of the final requirement is to prevent the reader from acting like Kurt Vonnegut's character Billy Pilgrim who could jump from one space-time location to another *(Slaughterhouse Five)*.

Taking a moment to describe how the final project will work will help a problem solver decide which ideas are important and which features are "icing on the cake." For this particular project, showing pictures, letting the reader quit and preventing time spasms are secondary features. They can await the solution of the central problem: procedures that will allow the reader to choose story branches and which do not use up Logo's memory space.

How Can Large Pieces of Text Be Presented Without Running Out of Memory?

The key is not to include the text inside Logo *procedures or variables*. Logo procedures and variables use up the Logo workspace. The alternative is to put the text in *file* on a disk. This file must be loaded into the computer in such a way that Logo does not act on the file in the same way that it acts when a file of procedures is loaded.

The Good News

It's possible to do this in most versions of Logo. If you are using Terrapin Logo, you should use the TEXTEDIT procedures on the Terrapin Utility Disk. These procedures allow you to create files of text and save them on the disk, read them from the disk and even print them. It's possible to do this in Apple Logo, but we don't know how. (We would appreciate hearing from a reader who can show us how to do this without using TEXTPRO from the Apple Logo Tool Kit. We would also like to hear how to do this in other versions of Logo.) It's even easier to do this with IBM Logo and Apple's new Apple Logo II. IBM Logo and Logo II have *primitives* that do what the rest of you will have to do with procedures.

Bare Bones Adventure Story Procedures

Suppose you have some text files stored on the disk, each file having one of the names used in the tree structure above. The last part of each of these story parts should tell the reader what to type in order to go to one of the available branches.

The following two procedures will let the reader use the computer to do the adventure (they are written in IBM Logo—which is almost the same as Apple Logo II):

```
TO ADVENTURE
  TEXTSCREEN
  SETWIDTH 80          Why not use 80 columns?
  CLEARTEXT            Start with a fresh screen.
  POFILE "INTRO        POFILE is the primitive that allows a
                       text file to be displayed but not put into
                       Logo's workspace. With Terrapin you
                       use a UTILITY procedure.
  GET.CHOICE           This subprocedure does all the rest of
                       the work. (See below.)
END

TO GET.CHOICE
  REPEAT 2 [ PRINT [ ] ]
  MAKE "CHOICE FIRST
    READLIST           This records the word that the reader
                       types in for the next choice.
  SETWIDTH 80
  CLEARTEXT
  POFILE :CHOICE       Whatever name the reader typed in
                       above will be the name of the file that
                       Logo will try to display. A recurring
                       process.
  GET.CHOICE
END
```

If you want to try out these bare bones procedures, you'll need to have some text files on the disk. We were able to use these procedures to read files that were typed with a word processor. But not all word processors produce text files. For example, *Bank Street Writer* files need to be converted into text files. Text files can be produced with the Logo editor. Some Logos have primitives that assist this effort. In the case of Terrapin, you use some of their TEXTEDIT procedures. If you have IBM Logo or Apple Logo II, the following procedure will work.

```
TO WRITE.PART
  TEXTSCREEN SETWIDTH 80
  PRINT [ WHAT IS THE
    NAME OF THIS STORY
    PART? ]
  MAKE "PART FIRST
    READLIST
  OPEN :PART           These two lines create the file on the
  CLOSE :PART          disk—with no text in it.
  EDITFILE :PART       This line re-opens the file and puts you
                       in the editor. When you press ESC, the
                       text is written onto the disk.
END
```

Here is an example of a file produced with this procedure.

```
?THIS IS THE BEGINNING OF THE LOGO CENTER ADVENTURE
STORY. WE ARE TYPING THIS IN THE LOGO EDITOR. YOU
GET INTO THE EDITOR BY USING THE COMMAND EDITFILE
FOLLOWED BY A FILE NAME. THIS ONLY WORKS AFTER
YOU HAVE CREATED A FILE USING THE OPEN AND THE
CLOSE COMMANDS. HERE ARE THE THREE LINES WE TYPED
IN ORDER TO GET INTO THE EDITOR:
  OPEN "INTRO
  CLOSE "INTRO
  EDITFILE "INTRO
ONCE YOU ARE IN THE EDITOR, YOU TYPE ALONG AS YOU
WOULD WITH AN OLD STYLE TYPEWRITER—THAT IS, WHEN
YOU GET NEAR THE END OF THE LINE, YOU PRESS THE
<ENTER> OR <RETURN> KEY TO SKIP DOWN AND BEGIN A
NEW LINE. THIS EDITOR LETS YOU USE ALL THE USUAL
EDITING FEATURES IN LOGO.
THESE BRANCHING ADVENTURE STORIES NEED TO END WITH
A CHOICE. THE CHOICE SHOULD BE PRESENTED IN THIS
FORM:
TYPE CAVE   IF YOU WISH TO ENTER THE MYSTERIOUS
            CAVE.
TYPE BEACH  IF YOU WANT TO EXPLORE THE BEACH.
TYPE QUIT   IF YOU WANT TO END THIS ADVENTURE.
```

Icing on the "Choose-Your-Own-Adventure" Cake

The GET.CHOICE procedure does not let anyone quit. It's easy to insert two lines that will check to see whether the reader typed in QUIT. You don't want the computer to go searching on the disk for a file called "QUIT. Instead, you want to throw the reader back to the top level of the computer. Here's how GET.CHOICE can be modified. (Some frills have been removed to help you focus better.)

```
TO GET.CHOICE
  MAKE "CHOICE FIRST READLIST
  TEST :CHOICE="QUIT
  IFTRUE [ THROW "TOPLEVEL ]
  POFILE :CHOICE
  GET.CHOICE
END
```

advertisement

BEST COPY AVAILABLE

This same approach can be used when the reader finally gets to one of the ending branches of the story. In the tree structure above, the branch labeled WEIGHT is a (sad) ending. Here is the weight chapter.

```
WEIGHT
WRONG MOVE. YE'

THE STEP CRUMBLES TO DUST. YOU LURCH BACKWARDS
AND FALL INTO THE DARKNESS BELOW. THE NEXT THING
YOU SEE IS WHITE MIST AND YOU HEAR BEAUTIFUL
MUSIC.

YOU RESOLVE THAT IN YOUR NEXT LIFE YOU WILL BE
MORE CAREFUL.
```

Can you figure out two lines to insert in GET.CHOICE that will do the following?

- Check to see whether WEIGHT was typed in;
- If it was, cause the WEIGHT file to be displayed;
- Throw the reader back to toplevel.

Can you figure out where to put these two lines?

Now let's tackle the problem of preventing the Billy Pilgrim phenomenon. Look back at the tree structure and suppose that after the introductory chapter you selected the CAVE option. Your screen reads:

```
CAVE
YOU HAVE ENTERED THE MYSTERIOUS CAVE WITH CREEPY
OLD SPIDER WEBS. IT IS DARK IN HERE. AS YOU MOVE
INTO THE CAVE, YOU ARE FORCED TO BEGIN CRAWLING
BECAUSE THE CEILING IS GETTING LOWER AND LOWER. AS
YOU ARE CRAWLING ALONG, YOU HEAR A HIGH-PITCHED
WHISTLING NOISE FROM TIME TO TIME AND FEEL AIR
RUSHING PAST YOUR EARS. LATER YOU WILL FIND OUT
THAT YOUR PARTNER HAS SEEN LARGE BATS FLYING FROM
THE NARROW CAVE INTO WHICH YOU CRAWLED.

YOU COME NOW TO A SPOT WHERE THE CAVE BECOMES
LARGER. YOU STAND AND LIGHT YOUR TRUSTY LAMP.
AHEAD OF YOU IS A LADDER THAT APPEARS TO GO UP TO
A BALCONY. TO YOUR RIGHT IS A WOODEN DOOR THAT
LOOKS AS IF IT IS ABOUT READY TO FALL APART.

TYPE 1          TO SEE A PICTURE OF THE DOOR.
TYPE 2          TO SEE A PICTURE OF THE LADDER.
TYPE LADDER     IF YOU DECIDE TO CLIMB THE LADDER.
TYPE DOOR       IF YOU DECIDE TO OPEN THE WOODEN
                DOOR
TYPE QUIT       IF YOU DECIDE TO QUIT.
```

If you were Billy Pilgrim, you could type BEACH and be quickly transported to the beach, even though going to the BEACH was not a legitimate option once you had entered the cave. Magic is fun, but it's nice to be able to control it.

Keeping Track of Where You Are in the Story

Suppose the computer could "know" which choices are legitimate at any given point in the story. Suppose further that the computer could keep track of the reader's place in the story. Then the computer could test to see whether the reader's choice was one of the legitimate choices. The computer needs a way to keep track of the information such as that shown in the following table.

CURRENT CHAPTER	LEGITIMATE CHOICES
INTRO	CAVE BEACH
CAVE	DOOR LADDER
BEACH	PART1 PART2 PART3
DOOR	PART16 PART17

A set of variables can be used to keep this information. These variables could be in a SETUP procedure that is included as a subprocedure of ADVENTURE, or they could be inserted near the beginning of the ADVENTURE procedure.

```
MAKE "INTRO [ CAVE BEACH ]
MAKE "CAVE [ DOOR LADDER ]
MAKE "BEACH [ PART1 PART2 PART3 ]
MAKE "DOOR [ PART16 PART17 ]
MAKE "LADDER [ WEIGHT SKIP ]
```

Sidetrip: Looking "Through" a Variable

If you need to learn more about Logo variables, take a few moments to try some exploring. Type in the five MAKE statements just presented. Then try these. When you think you've got the hang of it, try filling in the blanks before you press the ⟨RETURN⟩ key.

WHAT TO TYPE	WHAT HAPPENS?
PRINT FIRST :DOOR	_____
PRINT FIRST "DOOR	_____
PRINT THING "LADDER	_____
PRINT THING FIRST :INTRO	_____
PRINT THING LAST :INTRO	_____
PRINT FIRST THING LAST :INTRO	_____

The Logo primitive THING takes a word (a variable name) as its input. THING tells you what object has been associated with the word. (The association between word and object was accomplished when you typed in all those MAKE statements.) THING can act like a telescope and allow you to look through one variable and see what is associated with one of the items inside that variable. Thus, when you type

```
PRINT FIRST THING LAST :INTRO
```

the computer figures out the sentence this way:

The value of INTRO is the list	[CAVE BEACH]
The LAST :INTRO is the word	"BEACH
The THING associated with the word "BEACH is the list	[PART1 PART2 PART3]
The FIRST of the list above is	"PART1

Try a few more of your own to help you with this idea.

The five MAKE statements above store information about the legitimate choices at any spot in the story. Next, the computer needs a way of keeping track of where it is in the story. Let's use a variable called "CURRENT.CHAP. This variable will change each time the reader makes a new choice. When the story begins, the current chapter will be "INTRO., so the following statement is inserted into the ADVENTURE procedure.

```
MAKE "CURRENT.CHAP "INTRO
```

Just to make sure you are comprehending, what results from typing :CURRENT.CHAP? _____.

What if you type THING: CURRENT.CHAP?_____.

Let's express (in almost ordinary English) what needs to happen to prevent the Billy Pilgrim phenomenon. Suppose that :CURRENT.CHAP is "CAVE. The legitimate choices are DOOR and LADDER. The procedure GET.CHOICE is waiting for you to type in your :CHOICE. Suppose you type in BEACH. We want the GET.CHOICE procedure to do a test before it tries to read in the file containing the BEACH chapter. The procedure must find out whether your choice (BEACH) is a member of the group of legitimate choices for the current chapter (CAVE). In Logo vernacular this is expressed as:

```
TEST MEMBERP :CHOICE THING :CURRENT.CHAP
```

It can be translated as: "Is it true that the value of the variable with the name CHOICE is a member of the object associated with the variable named CURRENT.CHAP?"

Read through the following procedures and see if you can see how they work to prevent the reader from making illegitimate choices. The subprocedure AGAIN is used when a person makes a wrong choice. It prevents the GET.CHOICE procedure from continuing until the reader types in a legitimate answer. This infinitely patient procedure is given below.

```
TO ADVENTURE
  MAKE "INTRO [ CAVE BEACH ]
  MAKE "CAVE [ DOOR LADDER ]
  MAKE "BEACH [ PART1 PART2 PART3 ]
  MAKE "DOOR [ PART16 PART17 ]
  MAKE "LADDER [ WEIGHT SKIP ]
  TEXTSCREEN SETWIDTH 80
  CLEARTEXT
  MAKE "CURRENT.CHAP "INTRO
  POFILE "INTRO
  GET.CHOICE
END

TO GET.CHOICE
  MAKE "CHOICE FIRST READLIST
  TEST MEMBERP :CHOICE THING :CURRENT.CHAP
  IFFALSE [ AGAIN ]
  SETWIDTH 80
  CLEARTEXT
  POFILE :CHOICE
  MAKE "CURRENT.CHAP :CHOICE   (This line updates :CURRENT.CHAP once the reader has made a legitimate choice.)
END

TO AGAIN
  PRINT [ WHAT DO YOU WANT TO DO? ]
  PRINT SENTENCE THING :CURRENT.CHAP [ OR QUIT? ]
  MAKE "CHOICE FIRST READLIST
  IF MEMBERP :CHOICE :CURRENT.CHAP [ STOP ]
  AGAIN
END
```

Displaying Pictures From the Story

The procedures above can be modified so that picture files can be read in. We'll leave this as a problem for our readers to solve. One approach is to include information about pictures in the "Legitimate Choices" table. The MAKE statements collecting this information look like this:

```
MAKE "CAVE [ DOORPICT DOOR LADDER LADDERPICT ]
```

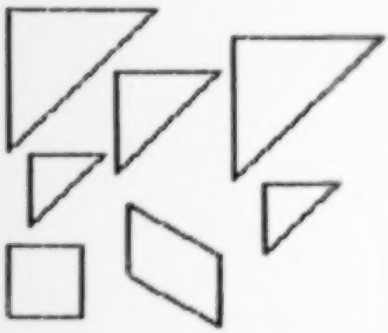
Can you put all this together now into your own Choose-Your-Own-Adventure project? Good luck! **END■**

[Kathleen Martin, Dept. of Education, University of Dallas, Dallas, TX 75061; Tim Riordon, Assoc. Dir. of Curr. and Trng., National Computer Training Institute, 39510 Paseo Padre Parkway, Fremont, CA 94556.]

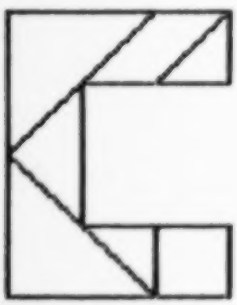
Kathleen Martin
and
Tim Riordon

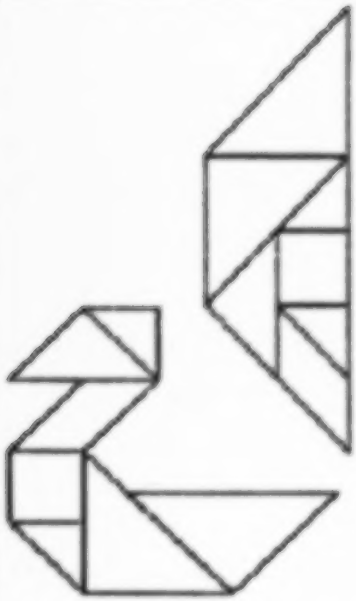
Tangrams and Turtles

Most kids like to put puzzles together. In this month's Logo Center we'll tackle the task of assembling puzzles using a collection of Logo procedures for various geometric shapes. Let's begin with a puzzle that is familiar to many teachers. It's the seven-piece Chinese puzzle called a tangram. We suggest that you start this project by giving each student an envelope containing the tangram pieces.



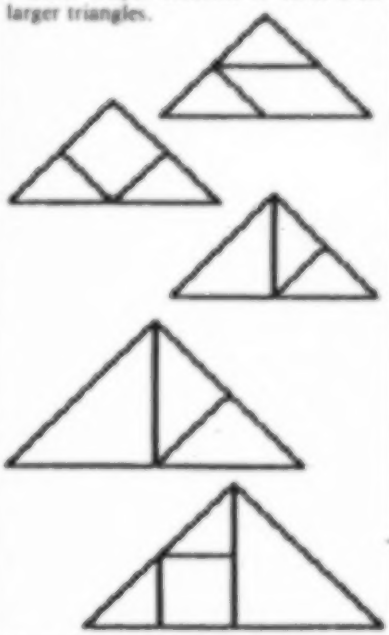
Instruct your students to assemble the seven pieces so that they form a single geometric shape. The possibilities are numerous, as seen in the puzzles below.





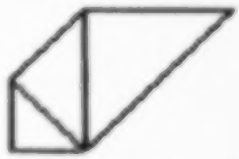
After your students have assembled a puzzle to their liking, take some time to discuss the relationships between the seven pieces. Look at the versatility of the two small triangles. They can combine to form a square, a parallelogram or a larger triangle.

Other shapes combine to form even larger triangles.



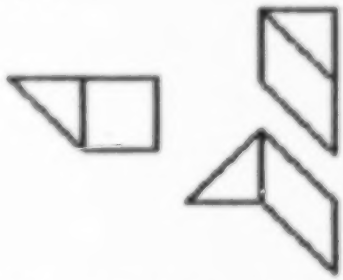
All of these triangles suggest the importance of this shape. Let's take a closer look at the three triangular pieces.

Move them together as shown below.



The longer side of the small triangle equals the shorter side of the medium triangle; its longer side, in turn, equals the shorter side of the large triangle.

What about the square and the parallelogram?



The shorter side of the small triangle is the same length as the side of the square and as the shorter side of the parallelogram. The longer side of the small triangle equals the longer side of the parallelogram.

Let's note at least one more important relationship among the triangles.

The longer side of the medium triangle is double the shorter side of the small triangle. The longer side of the largest triangle is double the longer side of the small triangle.

So what do all of these comparisons tell us? They tell us something quite important: Every side of each shape is equivalent to or a multiple of one of the sides of the small triangle. Once we have written a Logo procedure for the small

triangle, it will be simple to write procedures for the other shapes.

Now, what do we know about the small triangle? Let's fold the triangle along the line shown below so that the two shorter sides are atop one another.

We can see that the sides seem to be the same length and the angles seem to be the same measure. A triangle with two equal sides is called *isosceles*.

To find the measures of the turning angles, extend the sides in the direction that the Turtle will be traveling when drawing the triangle.

Since the Turtle turns through a total of 360 degrees in drawing any polygon, we know that the sum of angles 1, 2 and 3 is 360 degrees. Angle 1 is a right angle. Can you show that it is equivalent to the corner of a square? Angles 2 and 3 must total 270 degrees (360 – 90). Since they are equal, each of the angles measures 135 degrees.

Now we have enough information to write a Logo procedure to help us find the length of the longer side of the small triangle. This side of a right triangle is called the *hypotenuse*.

```
TO SM.RT.TRI :HYPOTENUSE
FD 30
RT 90
FD 30
RT 135
FD :HYPOTENUSE
RT 135
END
```

Through a process of trial and error, we can find the hypotenuse to be 42 turtlesteps. (The PRINT POS command can be used to estimate the hypotenuse even more closely.) The procedure for the small right triangle can be rewritten as follows:

```
TO SM.RT.TRI
FD 30
RT 90
FD 30
RT 135
FD 42
RT 135
END
```

Using the relationships discovered previously, procedures can also be written for the other geometric shapes.

```
TO MED.RT.TRI
FD 42
RT 90
FD 42
RT 135
FD 60
RT 135
END

TO LG.RT.TRI
FD 60
RT 90
FD 60
RT 135
FD 85
RT 135
END

TO SQUARE
REPEAT 4 [ FD 30 RT 90 ]
END

TO PARALLELOGRAM
REPEAT 2 [ FD 30 RT 135 FD 42
  RT 45 ]
END
```

The task now is to write a single procedure which assembles the shapes into the desired puzzle. Procedures for TRAPEZOID, DUCK and C are given below.

```
TO TRAPEZOID
PU SETX 100 LT 90 PD
LG.RT.TRI
PU FD 60 RT 45 FD 85 RT 135 PD
LG.RT.TRI
PU RT 135 FD 42 LT 180 PD
MED.RT.TRI
RT 45
PARALLELOGRAM
FD 30 RT 90
SM.RT.TRI
LT 90
SQUARE
FD 30 RT 45 FD 42 RT 135
SM.RT.TRI
END

TO DUCK
PU FD 80 RT 45 PD
MED.RT.TRI
FD 42 RT 45
SM.RT.TRI
PU RT 90 FD 30 LT 90 PD
PARALLELOGRAM
LT 45 BK 42 RT 45
SQUARE
LT 90 BK 30 RT 90
SM.RT.TRI
PU RT 45 FD 42 LT 45 FD 60
  RT 180 PD
LG.RT.TRI
RT 45 FD 60 RT 135 FD 85
  RT 135
LG.RT.TRI
END

TO C
PU BK 60 PD
SQUARE
PU LT 45 FD 42 RT 135 PD
SM.RT.TRI
LT 135
MED.RT.TRI
BK 42 LT 45
LG.RT.TRI
RT 45 FD 85 RT 45
PARALLELOGRAM
FD 30 RT 90
SM.RT.TRI
END
```

Students will need help in describing the moves between shapes. These moves usually involve traveling along the border of a previously drawn shape. There is no "correct" answer for the puzzle. The "best" answers, however, are those which are most economical; that is, those which use the fewest commands possible. With practice, students can become quite proficient at this activity. Initially, most students will want to work step by step in the draw mode. As they gain confidence and become more adept at judging moves, encourage them to use the edit mode, testing and modifying their procedures as needed.

If the right triangle seems too complex for your students, then design a puzzle based on simpler shapes. The one below uses an equilateral triangle as the fundamental shape. The hexagon and rhombus are related to the equilateral triangle in ways similar to the tangram shapes.

Assembling geometric puzzles using Logo procedures is quite challenging. To prevent the challenge from deteriorating

into tedium, it is important that students be allowed to select or to design a *puzzle* whose solution they are seriously interested in pursuing. As is always true with Logo, it is very important to distinguish when it is appropriate to intervene and give students a nudge and when it is better to allow them to struggle. More helpful hints may be needed with this task than with others you have attempted.

What About SQRT?

If you have sufficient time to devote to these puzzles, you may wish to introduce your students to the SQRT command. That introduction will not be provided here. (See references for appropriate activities to introduce SQRT.) Instead, we will simply present an alternative procedure with an abbreviated explanation.

When working with right triangles, the Pythagorean theorem can be especially helpful. In the case of the 45-45-90 tangram triangles, the Pythagorean theorem can be applied to find the hypotenuse of the triangle.

$$C^2 = A^2 + B^2$$

where C is the hypotenuse and A and B are the other two sides of the right triangle.

Since the 45-45-90 triangle is isosceles, the other two sides are equal. Therefore, A = B and

$$C^2 = 2A^2$$
$$C = \sqrt{2} \bullet A$$

Using SQRT, the procedure reads as follows:

```
TO RT.TRI :SIDE
FD :SIDE
RT 90
FD :SIDE
RT 135
FD :SIDE * (SQRT 2)
RT 135
END
```

RT.TRI 30 yields the same small triangle as the SM.RT.TRI procedure.

When your students use Logo procedures to explore tangrams and similar puzzles, they are learning a lot about the relationships among geometric shapes. SQRT is a powerful Logo command which can assist in that exploration if students understand its meaning and do not just use it as an inexplicable formula. **END■**

[Kathleen Martin, Dept. of Education, University of Dallas, Dallas, TX 75061; Tim Riordon, Assoc. Dir. of Curr. and Trng., National Computer Training Institute, 39510 Paseo Padre Parkway, Fremont, CA 94556.]

References

Seymour, Dale. *Tangrumath*. Palo Alto: Creative Publications, 1971.

Bearden, Donna, and Kathleen Martin, *Micrequests*, Vol. 1 No. 1, September 1984.

Kathleen Martin and Tim Riordon

Beyond Turtle Graphics: Data Bases with Logo

For the next three months, *The Logo Center* column will introduce readers to the concepts involved in data base programs. One of the important concepts is how to create and manipulate large amounts of information without using up Logo's workspace. The newer releases of Logo (IBM Logo, Apple Logo II and Macintosh Logo) make this task much easier, because these implementations include some primitives that otherwise need to be created. This series will include some suggestions for people who are still using Terrapin Logo or the original Apple Logo. The learning vehicle will be the creation of a FRIENDS data base. This column will discuss how Logo files are stored and accessed.

Below is a representation of four different files from the FRIENDS data base.

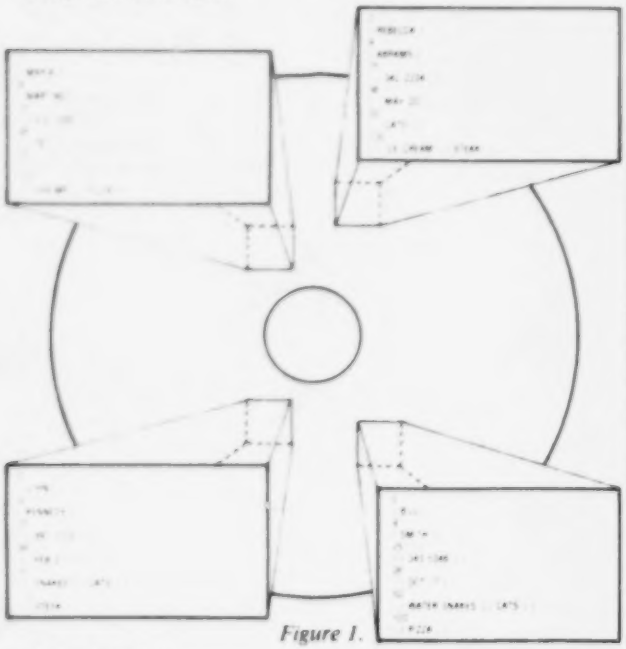


Figure 1.

Major Point 1

Notice that the information is stored as a series of lines. Each line is a list, or list of lists. Notice also that each line begins at a certain position in the file. Compare the "MARTINEZ file with the others and notice that the list of favorite foods *always* begins at position 100. This way of organizing the information makes it possible to tell the computer to go into a certain record file, go directly to a certain position in that file, and report back the information listed on that line in the record. The information we are going to store in our FRIENDS data base is always going to be organized in individual record files as follows:

- Position 0 will be the beginning of the list that contains the particular friend's **first name.** (Because the information is stored in **lists,** position 0 will be a left bracket. Position 1 will be the first letter of the name.) The two underscores after [MARIA] show that positions 7 and 8 are unoccupied.
- Position 9 will be a new line in the record and is the beginning of the list that holds the person's **last name.**
- Position 25 will be a new line and the beginning of the place where the telephone number is stored.
- Position 36 is where you will find your friend's birthday.
- Your friend's favorite pets are listed beginning at position 50.
- Your friend's favorite foods will be found beginning at position 100.

Once information has been stored this way, retrieving information (from the Kennedy file, for example) can be accomplished in this manner:

- Tell the computer to open up the "KENNEDY file.
- Tell the computer to pay attention to that file instead of to the keyboard.
- Go directly to position 36.
- Read the list of information stored at that position. (The information is [[FEB 2]].)

The newer implementations of Logo have several commands that help accomplish these tasks:

```
OPEN "MARTINEZ          SETREAD "MARTINEZ
SETREADPOS 100          READLIST
```

Here's how these commands work (shown with some different inputs). OPEN "MARTINEZ causes the drive to spin until the "MARTINEZ file is located. SETREAD "MARTINEZ causes the computer to direct its attention to this file instead of to the keyboard. SETREADPOS 100 causes the "reader" to position itself at position 100. READLIST causes the computer to output whatever list begins at position 100. In this case, READLIST outputs [[SHRIMP] [PIZZA]]. Now, test your understanding: What would the following sequences generate? Each example refers to one of the four data files shown above.

```
OPEN "ABRAMS
SETREAD "ABRAMS
SETREADPOS 0
READLIST
```

(result) _____

(a)

```
OPEN "SMITH
SETREAD "SMITH
SETREADPOS 9
MAKE "STORAGE READLIST
PRINT :STORAGE
```

(result) _____

(b)

```
MAKE "BUFFER "KENNEDY
OPEN "KENNEDY
SETREAD "KENNEDY
SETREADPOS 50
MAKE "BUFFER ( LPUT READLIST : BUFFER )
SHOW :BUFFER
```

(result) _____

(c)

BEST COPY AVAILABLE

(All answers appear at the end of the column.)

In the last case did you predict that SHOW :BUFFER would cause the screen to display

[KENNEDY [[SNAKES] [CATS]]] ?

The LPUT operation uses two inputs: an **object** of some type —in the above case [[SNAKES] [CATS]] ; and an **existing list**—in the above case [KENNEDY]. The result of the LPUT operation is to put the first object inside the existing list *in the last position* in the list.

Major Point 2

In the representation of how the files are stored on the disk (figure 1), the records do not appear to be placed on the disk in any particular order. They aren't ordered alphabetically, or by birthdate, or according to the number of pets a person has. But consider each of the lists associated with the following Logo variables:

```
MAKE "ALPHAINDEX [ ABRAMS KENNEDY MARTINEZ SMITH ]
```

Which record on the disk would be "considered" by the following command?

```
OPEN FIRST :ALPHAINDEX _____
```
(d)

```
MAKE "PHONEINDEX [ MARTINEZ ABRAMS SMITH KENNEDY ]
```

Study the telephone numbers of the four people to see why :PHONEINDEX lists the people in the particular order shown.

What information would be "read" by:

```
OPEN FIRST BUTFIRST :PHONEINDEX
SETREAD FIRST BUTFIRST :PHONEINDEX
SETREADPOS 50
READLIST _____
```
(e)

Many data base programs use indexes to control the order in which individual records are processed. The :ALPHAINDEX causes the "ABRAMS file to be processed first, then "KENNEDY, "MARTINEZ and finally "SMITH. But :PHONEINDEX causes the _____ file to be processed first.
(f)

To see if you understand this, finish the following statement and create an index that would cause records to be processed in the order of your friends' birthdays. Such an index would be useful if you wanted to print out a list of friends ordered by their birthdays.

```
MAKE "BIRTHINDEX _____
```
(g)

When files for new friends are added to your data base, the various indexes must be updated.

Here are answers for this month's column:

(a) [REBECCA]
(b) [SMITH]
(c) [KENNEDY [[SNAKES] [CATS]]]
(d) "ABRAMS
(e) [[CATS]] (from the ABRAMS file)
(f) MARTINEZ
(g) [KENNEDY ABRAMS SMITH MARTINEZ] END■

Kathleen Martin and Tim Riordon

Life Beyond Turtle Graphics: Data Bases With Logo Part 2

Moving to the Computer

Our programming goal this month is to develop some of the procedures needed to create your FRIENDS data base and to use the data base to retrieve information. Next month we will develop these ideas further. Eventually, your data base could have a MAIN MENU similar to this:

```
FRIENDS DATA BASE

1=ADDFRIEND
2=CHANGEINFO
3=USEINFO
4=PRINTINFO
5=QUIT
TYPE IN A NUMBER AND PRESS ENTER: __
```

Here is a diagram showing all the subprocedures called by any given procedure. For example, ADDFRIEND calls COLLECT and STORE.

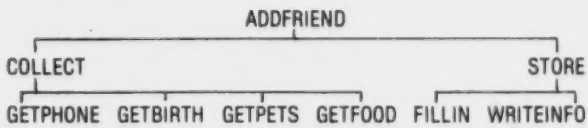


Figure 2.

Type in the following procedures and study the comments for each.

```
TO ADDFRIEND
COLLECT
STORE
END
```

The superprocedure for adding a new friend to your data base.

```
TO COLLECT
TS
CT
PRINT [ WHAT IS YOUR FRIEND'S FIRST NAME? ]
MAKE "FIRSTNAME READLIST
PRINT [ ]
PRINT [ WHAT IS YOUR FRIEND'S LAST NAME? ]
MAKE "LASTNAME READLIST
PRINT [ ]
MAKE "PHONE GETPHONE
PRINT [ ]
MAKE "BIRTHDAY GETBIRTH
PRINT [ ]
MAKE "PETS GETPETS
PRINT [ ]
MAKE "FOOD GETFOOD
MAKE "RECORDNAME FIRST :LASTNAME
MAKE "RECORD ( LIST :FIRSTNAME :LASTNAME :PHONE --
  :BIRTHDAY :PETS :FOOD )
END
```

COLLECT gathers all the information about one friend. Ultimately this information is stored in two variables—:RECORDNAME and :RECORD. The information is passed on and used by the STORE procedure.

Notice that each item of information—first name, last name, phone, etc.—is "remembered" with a variable while the COLLECT procedure is working. The content of each variable is a list or list of lists containing the specific items of information.

Thus, when you run ADDFRIEND and type in the information above for Bill Smith, COLLECT will create the following variables:

```
:FIRSTNAME is [ BILL ]                    :LASTNAME is [ SMITH ]
:PHONE is [ [ 343 5348 ] ]                :BIRTHDAY is [ [ OCT 17 ] ]
:PETS is [ [ WATER SNAKES ] [ CATS ] ]    :FOOD is [ [ PIZZA ] ]
:RECORD NAME is "SMITH
:RECORD is

 [ [ [ BILL ] [ SMITH ] ] [ [ 343 5348 ] ] [ [ OCT 17 ] ]  [ [ WATER SNAKES ] [ CATS ] ] [ [ PIZZA ] ] ]
```

The lists of information about phone, birthday, pets and food are created by the four procedures shown below:

```
TO GETPHONE
PRINT [ WHAT IS YOUR FRIEND'S PHONE NUMBER? ]
OUTPUT ( LIST READLIST )
END

TO GETBIRTH
PRINT [ WHEN IS YOUR FRIEND'S BIRTHDAY? ]
OUTPUT ( LIST READLIST )
END

TO GETPETS
PRINT [ HOW MANY PETS DOES YOUR FRIEND HAVE? ]
MAKE "NUM FIRST READLIST
MAKE "PETS [ ]
REPEAT :NUM [ PRINT [ TYPE IN A PET AND PRESS —
  ENTER ] MAKE "PETS LPUT READLIST :PETS PRINT [ ] ]
OUTPUT :PETS
END

TO GETFOOD
PRINT [ WHAT ARE YOUR FRIEND'S 2 FAVORITE FOODS? ]
MAKE "FOOD [ ]
REPEAT 2 [ PRINT [ TYPE IN A FOOD AND PRESS —
  ENTER ] MAKE "FOOD ( LPUT READLIST :FOOD ) ]
OUTPUT :FOOD
END
```

Each of these procedures prints a message telling the person what kind of information to enter. Whatever a person types in is stored as a list.

In the cases of pets and foods, you can put in information about more than one food or pet.

Notice that each procedure OUTPUTs a list. The lists created are used as contents of variables in the COLLECT procedure.

Which variable in the COLLECT procedure is created with the help of GETFOOD? _____
(h)

(This and other answers appear at the end of the column.)

The next process is for the computer to store this information about a single friend on the disk. The reason for storing the information on the disk is that the computer's memory is relatively limited, especially Logo's memory! The STORE procedure and its two helpers, FILLIN and WRITEINFO, accomplish this task. Learning to use Logo's file-handling capability is important for Logo users because it opens up many new possibilities for using Logo.

```
TO STORE
OPEN :RECORDNAME
SETWRITE :RECORDNAME
FILLIN :RECORDNAME 101
WRITEINFO :RECORD [ 0 9 25 36 50 100 ]
CLOSE :RECORDNAME
SETWRITE "CON
END
```

STORE orchestrates the process of storing the information on the disk. OPEN "SMITH begins the process of creating a disk file. SETWRITE "SMITH causes commands such as TYPE or PRINT to print the information onto the disk instead of on the screen. The last line of STORE redirects printing to the screen, archaically named "CON for console.

```
TO FILLIN :FILE :LENGTH
MAKE "SPACE :LENGTH — FILELEN :FILE
IF :SPACE > 0 [ REPEAT :SPACE [ TYPE CHAR 32 ] ]
END
```

The purpose of FILLIN is to create a blank record at least 100 bytes (spaces) long. When you have all these procedures working, remove the line

FILLIN :RECORDNAME 101

from STORE and see what happens.

```
TO WRITEINFO :RECORD :FIELDPOSITIONS
IF :RECORD=[ ] [ STOP ]
SETWRITEPOS FIRST :FIELDPOSITIONS
PRINT FIRST :RECORD
WRITEINFO BUTFIRST :RECORD BUTFIRST :FIELDPOSITIONS
END
```

WRITEINFO is a little recursive procedure that does all the labor. The important point to notice is that this procedure always writes a given category of information at the same beginning position. The two inputs to WRITEINFO are lists. :RECORD contains six items: [BILL] [SMITH] [[343 5348]] etc. WRITEINFO moves the disk file writer to position 0 and prints the list [BILL]. The recurring process then moves the disk file writer to position 9 and prints [SMITH]. The third copy of the procedure moves the writing head to position 25 and writes _____.

(i)

Recall the earlier explanation about always positioning a certain type of information in the same place in each file. By doing this, the computer can find the information more quickly, because it does not have to search through each disk file from the beginning.

If you wanted to set up your files so that information about a person's birthday always occurred at position 40, how would you change the inputs to WRITEINFO?

WRITEINFO :RECORD [_____]

(j)

Run the ADDFRIEND procedure five or six times to create some data files. When you have finished, you will need to create an alphabetical index. If your friends' names were Smith, Abrams and Kennedy, you would create this index by typing MAKE "ALPHAINDEX [ABRAMS KENNEDY SMITH]. Notice that it is not necessary that you *enter* your friends into the data base in any particular order. The various indexes you create will take care of ordering the records for processing.

Using Your FRIENDS Data Base

Examine the following menu screen, produced by the procedure listed below.

```
THERE ARE THREE WAYS YOU CAN USE YOUR FRIENDS
DATA BASE

1  FIND A SPECIFIC PIECE OF INFORMATION ABOUT A
   SPECIFIC FRIEND. FOR EXAMPLE . . . WHEN IS MARY
   SMITH'S BIRTHDAY?

2  LIST A SPECIFIC PIECE OF INFORMATION ABOUT EACH
   FRIEND. FOR EXAMPLE . . . LIST THE FAVORITE FOODS
   OF EACH FRIEND.

3  SEARCH FOR CERTAIN INFORMATION. FOR EXAMPLE . . .
   DO ANY OF MY FRIENDS LIKE PIZZA?

TYPE THE NUMBER FOR THE WAY YOU WOULD LIKE TO USE
YOUR DATA BASE.
```

```
TO USEINFO
TEXTSCREEN CLEARTEXT
SETUP
PRINT [ THERE ARE THREE WAYS YOU CAN USE YOUR ]
PRINT [ FRIENDS DATA BASE ]
PRINT [ ]
PRINT [ 1  FIND A SPECIFIC PIECE OF INFOR- ]
PRINT [    MATION ABOUT A SPECIFIC FRIEND. ]
PRINT [    FOR EXAMPLE . . . WHEN IS MARY ]
PRINT [    SMITH'S BIRTHDAY? ]
PRINT [ ]
PRINT [ 2  LIST A SPECIFIC PIECE OF INFOR- ]
PRINT [    MATION ABOUT EACH FRIEND. ]
PRINT [    FOR EXAMPLE . . . LIST THE FAVORITE ]
PRINT [    FOOD OF EACH FRIEND. ]
PRINT [ ]
PRINT [ 3  SEARCH FOR CERTAIN INFORMATION. ]
PRINT [    FOR EXAMPLE . . . DO ANY OF MY ]
PRINT [    FRIENDS LIKE PIZZA? ]
PRINT [ ]
PRINT [ TYPE THE NUMBER FOR THE WAY YOU WOULD ]
PRINT [ LIKE TO USE YOUR DATA BASE. ]
MAKE "CHOICE READWORD
IF :CHOICE="1 [ GETINFO ]
IF :CHOICE="2 [ FINDALL ]
IF :CHOICE="3 [ SEARCH ]
END
```

The project we are developing will eventually perform the functions shown above. Many procedures are involved in accomplishing these three different jobs. The diagram in figure 3 shows the various subprocedures called by any given procedure.

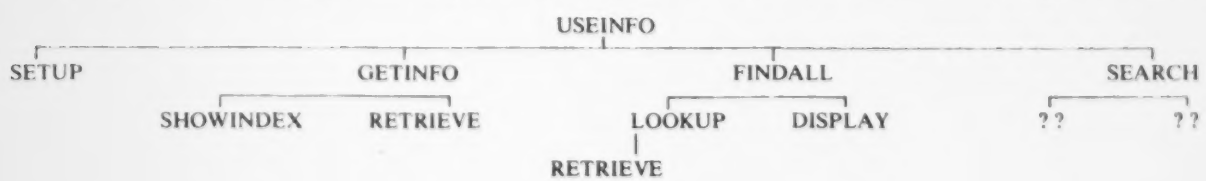


Figure 3.

Type in the procedures below. (The procedures used in the SEARCH routine will be presented next month. But perhaps you can write them before the column appears.)

Finding Information About a Specific Friend

```
TO SETUP
ERNS
MAKE "ALPHAINDEX [ _ _?_ _ _ _?_ _ _ _?_ _ _ _?_ _ ]
MAKE "FIRSTNAME 0
MAKE "LASTNAME 9
MAKE "PHONE 25
MAKE "BIRTHDAY 36
MAKE "PETS 50
MAKE "FOOD 100
MAKE "BUFFER [ ]
MAKE "STORAGE [ ]
MAKE "SEARCHSTORAGE [ ]
END
```

SETUP creates some variables that are used in the USEINFO routine. "BUFFER is the name of the variable that temporarily holds information in the GETINFO sequence. Just before GETINFO ends, the last line tells the computer to PRINT the contents of the "BUFFER variable.

"STORAGE is the name of the variable that temporarily holds information for the FINDALL sequence. Since this routine is designed to print out information for every person in your FRIENDS data base, the contents of "STORAGE could look like this just before the contents are exhibited by the DISPLAY procedure:

:STORAGE could be

[[ABRAMS [[STEAK]]] [KENNEDY [[ICE CREAM] [PIZZA]]] [SMITH [[PIZZA]]]]

Believe it or not, this list contains three items—each item is a list pairing up a person's name with all the foods the person likes. The underlining above helps to show this structure. This list is presented on the screen by the display procedure—but in a more readable format.

```
TO GETINFO
CT
PRINT [ YOU CAN FIND OUT ABOUT THE FOLLOWING →
  FRIENDS ]
PRINT [ ]
SHOWINDEX :ALPHAINDEX
PRINT [ ]
PRINT [ WHICH FRIEND DO YOU WANT TO FIND OUT →
  ABOUT? ]
MAKE "RECORDNAME READWORD
PRINT [ ]
PRINT [ YOU CAN FIND OUT YOUR FRIEND'S: ]
PRINT [      FIRST NAME ]
PRINT [      LAST NAME ]
PRINT [      PHONE NUMBER ]
PRINT [      BIRTHDAY ]
PRINT [      FAVORITE PETS ]
PRINT [      FAVORITE FOODS ]
PRINT [ TYPE IN ONE OF THE FOLLOWING WORDS: ]
PRINT [ ]
PRINT [ FIRSTNAME LASTNAME PHONE BIRTHDAY PETS →
  FOOD ]
MAKE "FIELDNAME READWORD
MAKE "FIELDPOSITION THING :FIELDNAME
CT
TYPE :RECORDNAME
REPEAT 8 [ TYPE CHAR 32 ]
PRINT RETRIEVE :RECORDNAME :FIELDPOSITION
END
```

The purpose of the GETINFO procedure is to find out something specific about a particular friend. Notice how this procedure causes all the names on file to be shown on the screen. This is the job of the SHOWINDEX procedure. It works provided you have remembered to update this index.

Near the end of the procedure, two variables are created. These are passed on and used by the procedure RETRIEVE, which does all the work.

When RETRIEVE finishes, it sends back a variable to GETINFO containing the information being sought. Which procedure causes the information to be printed on the screen?

(k)

```
TO SHOWINDEX :INDEX
IF :INDEX=[ ] [ STOP ]
PRINT FIRST :INDEX
SHOWINDEX BUTFIRST :INDEX
END
```

```
TO RETRIEVE :RECORDNAME :FIELDPOSITION
OPEN :RECORDNAME
SETREAD :RECORDNAME
SETREADPOS :FIELDPOSITION
MAKE "BUFFER READLIST
SETWRITE "CON
CLOSE :RECORDNAME
OUTPUT :BUFFER
END
```

RETRIEVE does all the hard work! It opens the file, puts the disk reader at the correct position to find the information that has been requested, and "remembers" information by storing it in a variable named "BUFFER.

After you have these procedures typed, test them by trying to get some specific information about one of your friends. Debug these procedures before going on to the next set.

Listing Information About All Your Friends

The second way that you might want to use your data base is to create a list that displays a certain category of information for all your friends. The following three procedures accomplish the task.

```
TO FINDALL
CT
PRINT [ WHAT WOULD YOU LIKE TO FIND OUT ABOUT
  ALL YOUR FRIENDS? ]
PRINT [ ]
PRINT [ YOU CAN FIND OUT YOUR FRIENDS': ]
PRINT [      FIRST NAMES ]
PRINT [      LAST NAMES ]
PRINT [      PHONE NUMBERS ]
PRINT [      BIRTHDAYS ]
PRINT [      FAVORITE FOODS ]
PRINT [      FAVORITE PETS ]
PRINT [ ]
PRINT [ TYPE IN ONE OF THE FOLLOWING WORDS: ]
PRINT [ ]
PRINT [ FIRSTNAME LASTNAME PHONE BIRTHDAY PETS —
  FOOD ]
MAKE "FIELDNAME READWORD
MAKE "FIELDPOSITION THING :FIELDNAME
LOOKUP :ALPHAINDEX :FIELDPOSITION
CLEARTEXT
DISPLAY :STORAGE
END
```

FINDALL asks you to type in a category of information.

Whatever you type in is stored in the variable named "FIELDNAME. This occurs as a result of the line

MAKE "FIELDNAME READWORD

If you type in PHONE, then :FIELDNAME is "PHONE.

Now consider the next line,

MAKE "FIELDPOSITION THING :FIELDNAME

This line transforms the field name you type in into the **position** number where that particular information is stored on the disk. For example, if :FIELDNAME is "PHONE, then :FIELDPOSITION is **the thing associated with the name "PHONE,** which is 25. This equation of a name with a position number was accomplished earlier in the SETUP procedure. Look back at the setup procedure and notice the series of MAKE statements that associate names with numbers. In the SETUP procedure, which statement sets up an association for the name "PETS? _____

(1)

```
TO LOOKUP :INDEX :FIELDPOSITION
IF :INDEX=[ ] [ STOP ]
MAKE "STORAGE ( LPUT ( LIST FIRST :INDEX RETRIEVE FIRST :INDEX :FIELDPOSITION ) :STORAGE )
LOOKUP BUTFIRST:INDEX :FIELDPOSITION
END
```

Let's explain the LOOKUP procedure by reference to a specific example. The purpose of LOOKUP is to gather and store information in a variable named "STORAGE. LOOKUP processes the records in the order of the particular index given as an input to the LOOKUP procedure. LOOKUP selects the first record in the index list and then calls on RETRIEVE to get a particular item of information from that record. (Later, another procedure—DISPLAY—will dump out the contents of the :STORAGE variable on the screen.) Figure 4 below is a diagram showing how LOOKUP, a recursive procedure, generates five

copies of itself. Notice that as each copy runs, the contents of the variable named "STORAGE grows larger. When LOOKUP begins, :STORAGE is just the empty list created earlier by the SETUP procedure. LOOKUP needs two inputs—a list of names to look up, and a position in the disk file where a particular type of information is to be found. In the example, the position number 25 is where phone number information is stored.

```
LOOKUP [ ABRAMS KENNEDY MARTINEZ SMITH ] 25
IF [ ABRAMS KENNEDY MARTINEZ SMITH ] = [ ] [ STOP ]
MAKE "STORAGE ( LPUT ( LIST "ABRAMS RETRIEVE –
  "ABRAMS 25 ) [ ] )
LOOKUP BUTFIRST [ ABRAMS KENNEDY MARTINEZ
  SMITH ] 25
END
```

This is what FINDALL starts running.
This copy of LOOKUP continues to run.
The contents of the variable named "STORAGE are now
[[ABRAMS [[342 2224]]]]

```
 LOOKUP [ KENNEDY MARTINEZ SMITH ] 25
 IF [ KENNEDY MARTINEZ SMITH ] = [ ] [ STOP ]
 MAKE "STORAGE ( LPUT ( LIST "KENNEDY RETRIEVE –
   "KENNEDY 25 ) [ ] )
 LOOKUP BUTFIRST [ KENNEDY MARTINEZ SMITH ] 25
 END
```

This copy is generated by the previous copy.
This copy of LOOKUP continues to run.
The contents of the variable named "STORAGE are now
[[ABRAMS [[342 2224]]] [KENNEDY [[397 1123]]]]

```
  LOOKUP [ MARTINEZ SMITH ] 25
  IF [ MARTINEZ SMITH ] = [ ] [ STOP ]
  MAKE "STORAGE ( LPUT ( LIST "MARTINEZ RETRIEVE –
    "MARTINEZ 25 ) [ ] )
  LOOKUP BUTFIRST [ MARTINEZ SMITH ] 25
  END
```

This copy is generated by the previous copy.
This copy LOOKUP continues to run.
The contents of the variable named "STORAGE are now
[[ABRAMS [[342 2224]]] [KENNEDY [[397 1123]]] –
[MARTINEZ [[312 1000]]]]

```
   LOOKUP [ SMITH ] 25
   IF [ SMITH ] = [ ] [ STOP ]
   MAKE "STORAGE ( LPUT ( LIST "SMITH RETRIEVE –
     "SMITH 25 ) [ ] )
   LOOKUP BUTFIRST [ SMITH ] 25
   END
```

This copy is generated by the previous copy.
This copy LOOKUP continues to run.
The contents of the variable named "STORAGE are now
[[ABRAMS [[342 2224]]] [KENNEDY [[397 1123]]] –
[MARTINEZ [[312 1000]]] [SMITH [[343 5348]]]]

```
    LOOKUP [ ] 25
    IF [ ] = [ ] [ STOP ]
    MAKE "STORAGE ( LPUT ( LIST "RETRIEVE " 25 ) [ ] )
    LOOKUP BUTFIRST [ ] 25
    END
```

This unusual copy is generated by the previous copy.
But it doesn't continue to run.
The contents of the variable named "STORAGE are just the same as they were before.

Figure 4.

At this point, all the copies of LOOKUP end. LOOKUP has completed the task of storing all the information about each friend's phone number. Control reverts back to the FINDALL procedure. The screen is cleared and the contents of the variable named "STORAGE are displayed in a more readable format by the DISPLAY procedure.

```
TO DISPLAY :LIST
IF :LIST=[ ] [ STOP ]
TYPE ITEM 1 FIRST :LIST
REPEAT 8 [ TYPE CHAR 32 ]
PRINT ITEM 2 FIRST :LIST
DISPLAY BUTFIRST :LIST
END
```

Preparing for Next Month

At this stage the FRIENDS data base is quite crude. Here are some enhancements you might work on:

- A superprocedure that creates the main menu shown near the beginning of the column and branches to the menu choice selected.
- Change the menus so that selections are made *either* by moving the cursor and pressing a key *or* by typing in a number.
- Create an UPDATE procedure that updates indexes. (You might want to have a disk file that contains each index.)
- You might develop a way to give a person more control over the format of a report being produced.
- Add error trapping to the routines so that incorrect responses don't cause havoc.

The Sad Truth for Users of Apple Logo I and Terrapin Logo

Often, the best way to stimulate progress is to say that something cannot be done. After many frustrating hours of high-order nerdism, several long distance phone calls to other Logo experts and the agony of several times feeling that we were so close, we've given up hope that a productive data base

can be created with these implementations of Logo. There just does not seem to be enough memory. For those who wish to follow this trail of tears, we'll summarize our attempt.

First, remember that the goal is to create a program that allows the information to be stored in files on the disk. With both Terrapin and Apple Logo I this is possible. Terrapin users should employ the TEXTEDIT procedures provided on the Terrapin Utility program. Our problems with Terrapin arose when we tried to supplement these procedures with others in order to do various kinds of searching and sifting of the information. We just kept running out of space.

The approach we tried with Apple Logo I involved storing all the information for a given person in a variable whose content was a list of six items. The first item was the person's first name, the second item was the person's last name, the third was a list containing the person's phone number, etc. Here is an example:

```
[ [ REBECCA ]  [ ABRAMS ] ] [ [ 342 2224 ] ]  [ [ MAY 20 ] ]
[ [ CATS ] ]  [ PIZZA ] ]
```

This variable—called "RECORD—was saved (using the SAVE command) in a file named "ABRAMS. Because we had first BURIED all the data base subprocedures, no procedures were saved in the "ABRAMS file—just the variable named "RECORD, which contained all the information listed above. To accomplish this with Apple Logo I, you can make the following modifications to the ADDFRIEND subprocedures:

```
ADDFRIEND   (same)
COLLECT     (same, but add the line ERNS near the beginning)
GETPHONE    (same)
GETBIRTH    (same)
GETPETS     (same)
GETFOOD     (same)
FILLIN      (not needed)
WRITEINFO   (not needed)
STORE       (modified as follows)
```

```
TO STORE
ERN [ FIRSTNAME LASTNAME PHONE BIRTHDAY PETS FOOD
  NUM ]
SAVE :RECORDNAME
ERN "RECORDNAME
RECYCLE
END
```

Type in these procedures, test them, erase all names and then bury them all. The procedures will let you create files on the disk that store information about your friends.

The information is packed into a list of lists. Then it should be easy to extract this information using the ITEM command, right? After all, you can find a person's food preferences by going through the following process:

1. LOAD the person's file. (This loads in the variable named "RECORD which contains that person's information.)
2. Extract ITEM 6 "RECORDNAME. (Since the favorite food list is always the sixth item in a record.)
3. Display the information, or store it for later display.

The problem announced itself with a strange error message:

```
LOAD DIDN'T OUTPUT TO OUTPUT IN LOAD "RECORDNAME
IN RETRIEVE.
```

We called several friends but no one could explain why that error message should have appeared in the context that it did.

Here are the subprocedures we tried to use:

```
USEINFO     (same)
GETINFO     (same)
SHOWINDEX   (same)
LOOKUP      (same)
DISPLAY     (same)
SEARCH      (OUT OF SPACE! We were pushing the space
            limit already.)
SETUP       (modified as follows)
RETRIEVE    (modified as follows)
```

```
TO SETUP
ERNS
RECYCLE
MAKE "FIRSTNAME 1
MAKE "LASTNAME 2
MAKE "PHONE 3
MAKE "BIRTHDAY 4
MAKE "PETS 5
MAKE "FOOD 6
MAKE "BUFFER [ ]
MAKE "STORAGE [ ]
END
```

```
TO RETRIEVE :RECORDNAME :FIELDPOSITION
LOAD :RECORDNAME
MAKE "BUFFER ITEM :FIELDPOSITION :RECORD
ERN [ RECORDNAME FIELDNAME FIELDPOSITION ]
ERN "RECORD
OUTPUT :BUFFER
END
```

When we changed the RETRIEVE procedure so that it was not a procedure that outputs, the error message went away. For some strange reason, LOAD did not work in a procedure that outputs. When we tried to rewrite the various USEINFO subprocedures in such a way that LOAD didn't occur in a procedure that "outputted" (argh! is there such a word?), we ran out of space.

If you want to do real data base work with Logo, get IBM Logo or Apple Logo II. (Of course, readers are welcome to prove us wrong!) **END■**

Here are the answers to the questions that have appeared throughout this month's column:

(h) "FOOD or :FOOD
(i) [[343 5348]]
(j) WRITEINFO :RECORD [0 9 25 40 50 100]
(k) PRINT RETRIEVE :RECORDNAME :FIELDPOSITION
(l) MAKE "PETS 50

[Tim Riordon, Assoc. Dir. of Curr. and Trng., National Computer Training Institute, 39510 Paseo Padre Parkway, Fremont, CA 94556.]

The Logo Center

Kathleen Martin and Tim Riordon

Life Beyond Turtle Graphics: Data Bases with Logo

This month we present the third and final part of this series on creating a data base using Logo. In the December/January issue, the basic concepts involved in the creation of a FRIENDS DATA BASE were introduced without any Logo procedures. The data base project we have asked our readers to embark upon uses disk files to store information. This approach permits large quantities of information to be processed without using memory-hungry Logo procedures, variables or property lists to store the information. In the February issue we introduced and explained a number of bare bones procedures. These procedures were sufficient to allow the data files for each friend to be created. A USEINFO procedure (with many attendant subprocedures) was also presented. This month we will modify some of the procedures already presented and add some new ones. We think this will leave you, the reader, with sufficient material to continue developing the project. You can begin to organize your thinking by examining the following diagrams:

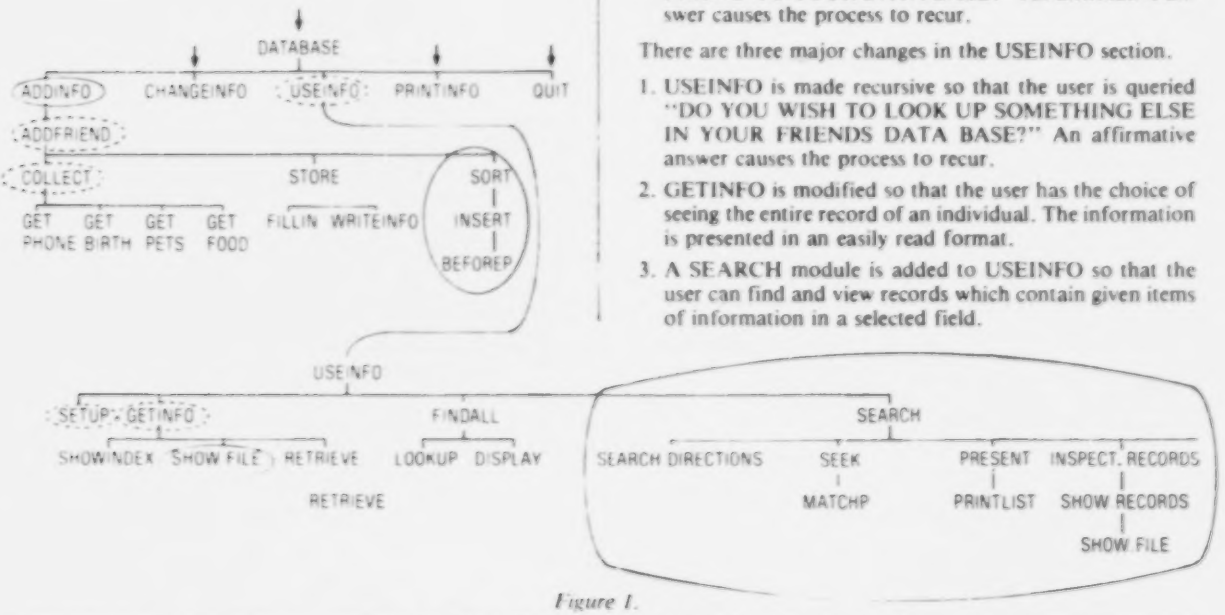


Figure 1.

Figure 1 shows the hierarchy of procedures. For example, the five procedures ADDINFO, CHANGEINFO, USEINFO, PRINTINFO and QUIT are all subprocedures of DATABASE. They are listed from left to right in the order that they would appear in the DATABASE. It is important to remember that a given procedure—DATABASE, for example—does not necessarily *call* upon all its subprocedures to perform. The PRINTINFO procedure is not called upon if the user does not select this alternative. How many subprocedures are potentially called by the COLLECT Procedure? _____(m) What is the first subprocedure that USEINFO calls? _____(n)

(Answers appear on p. 72.)

- Procedures and sections surrounded with a dotted line were introduced in Part 2 and will be modified below.
- Procedures that have been circled are new this month.
- Procedures with an arrow pointing to them have not been developed in this series. They are left for you to puzzle over.
- Remaining procedures were presented last month and are not modified this month.

This month's enhancement to the FRIENDS DATA BASE project falls into two broad areas. The program section that takes care of adding friends to the data base is improved in two ways:

1. The process of updating an index is automated so that when a new batch of friends is added to the data base, the ALPHAINDEX is automatically sorted and stored. Recall that the ALPHAINDEX is a variable whose value is the list of all the record names in the data base. This list is alphabetized. Review Part 1 of the series if you cannot recall the purpose of an index. The ALPHAINDEX is used by some of the procedures in the USEINFO module and it causes _____.(o)
2. The ADDFRIEND procedure is made recursive and the user is queried, "DO YOU WANT TO ADD ANOTHER FRIEND TO YOUR DATA BASE?" An affirmative answer causes the process to recur.

There are three major changes in the USEINFO section.

1. USEINFO is made recursive so that the user is queried "DO YOU WISH TO LOOK UP SOMETHING ELSE IN YOUR FRIENDS DATA BASE?" An affirmative answer causes the process to recur.
2. GETINFO is modified so that the user has the choice of seeing the entire record of an individual. The information is presented in an easily read format.
3. A SEARCH module is added to USEINFO so that the user can find and view records which contain given items of information in a selected field.

BEST COPY AVAILABLE

Improvements to the ADDINFO Section

Ideally, the process of maintaining an alphabetical index of records should happen automatically each time you finish adding a batch of friends to your data base. A variable named "NEWBATCH is initialized before you start the ADDFRIEND procedure. The ADDFRIEND procedure calls COLLECT, a procedure that gathers information and stores it all temporarily as variables until it is time to write the information in a file on the disk. A single new line has been added at the end of the COLLECT procedure

```
MAKE "NEWBATCH LPUT :RECORDNAME :NEWBATCH
```

As you add several new friends, their names are inserted in the list associated with the name "NEWBATCH. Suppose you have just added JONES and SMITH to your data base and you are now adding information for your friend Sarah Taylor. What will :NEWBATCH be when you have finished? _____ _____(p)

Three procedures are needed to take the new batch of names and insert them into the existing index in alphabetical order. Type in these three procedures and then test them by typing

```
SORT [ BETTY BET ABE WILLIAM MARY ] [ ALL CARLA
JIM NELSON SMITH ]

TO SORT :NEWBATCH :INDEX
IF EMPTYP :NEWBATCH [ OUTPUT :INDEX ]
MAKE "INDEX INSERT FIRST :NEWBATCH :INDEX
OUTPUT SORT BUTFIRST :NEWBATCH :INDEX
END

TO INSERT :OBJECT1 :LIST
IF EMPTYP :LIST [ OUTPUT ( LIST :OBJECT1 ) ]
IF BEFOREP :OBJECT1 FIRST :LIST [ OUTPUT FPUT -
  :OBJECT1 :LIST ]
OUTPUT FPUT FIRST :LIST INSERT :OBJECT1 BUTFIRST :LIST
END

TO BEFOREP :WORD1 :WORD2
IF :WORD1="  [ OUTPUT "TRUE ]
IF :WORD2="  [ OUTPUT "FALSE ]
IF ASCII FIRST :WORD1 < ASCII FIRST :WORD2 -
  [ OUTPUT "TRUE ]
IF ASCII FIRST :WORD1 > ASCII FIRST :WORD2 -
  [ OUTPUT "FALSE ]
OUTPUT BEFOREP BUTFIRST :WORD1 BUTFIRST :WORD2
END
```

The SORT procedure orchestrates the process. It takes each item, one by one, from the first list *and then tells* INSERT to put the item into the second list in the proper place. When there are no more items in the first list, SORT stops and outputs the second list, which is now larger than it was before the whole operation began.

INSERT is the real workhorse. Readers wishing to understand how such a short procedure can accomplish so much work are urged to "map out" the recursion using the technique described in "Teaching Recursion to Students," *The Computing Teacher*, November-February, 1983-84.

The BEFOREP procedure (a primitive in Apple Logo 2) outputs either a "TRUE or a "FALSE depending upon whether :WORD1 comes before :WORD2.

If you were going to create an index organized by month and date of birth, you would need to create a different procedure and use it in INSERT in place of BEFOREP.

If you had fields of numerical information in your data base (for example, how many times your friends have moved), Logo provides a primitive to use in the INSERT procedure that will sort the list in numerical order. It is the < operator. Rewrite TO INSERT as follows:

```
TO INSERT :NUM :NUMLIST
IF EMPTYP :NUMLIST [ OUTPUT ( LIST :NUM) ]
IF :NUM < FIRST :NUMLIST [ OUTPUT FPUT :NUM -
  :NUMLIST ]
OUTPUT FPUT FIRST :NUMLIST INSERT :NUM BUTFIRST -
  :NUMLIST
END
```

Then try PRINT SORT [17 45 23] [10 20 30 40]. Your result should be [10 17 20 23 30 40 45].

Now examine the three procedures ADDINFO, the recursive ADDFRIEND procedure and the slightly revised COLLECT procedure. (The check marks [✓] call your attention to any lines that have been added or revised in procedures presented last month.)

```
TO ADDINFO
LOAD "INDEXES
MAKE "NEWBATCH []
ADDFRIEND
MAKE "ALPHAINDEX SORT :NEWBATCH :ALPHAINDEX
PPROP "ALPHAINDEX "VALPKG "INDEXES
CLEARTEXT
PRINT [ THE COMPUTER IS NOW SAVING THE UPDATED -
  INDEXES ]
ERASEFILE "INDEXES
SAVE "INDEXES "INDEXES
END

TO ADDFRIEND
COLLECT
STORE
✓ PRINT [] PRINT []
✓ TYPE [ DO YOU WANT TO ADD ANOTHER FRIEND TO -
    YOUR DATA BASE? ]
✓ MAKE "REPLY READWORD
✓ IF FIRST :REPLY="Y [ ADDFRIEND ]
END

TO COLLECT
TS
CT
PRINT [ WHAT IS YOUR FRIEND'S FIRST NAME? ]
MAKE "FIRSTNAME READLIST
PRINT []
PRINT [ WHAT IS YOUR FRIEND'S LAST NAME? ]
MAKE "LASTNAME READLIST
PRINT []
MAKE "PHONE GETPHONE
PRINT []
MAKE "BIRTHDAY GETBIRTH
PRINT []
MAKE "PETS GETPETS
PRINT []
MAKE "FOOD GETFOOD
MAKE "RECORDNAME FIRST :LASTNAME
MAKE "RECORD ( LIST :FIRSTNAME :LASTNAME :PHONE -
  :BIRTHDAY :PETS :FOOD
✓ MAKE "NEWBATCH LPUT :RECORDNAME :NEWBATCH
END
```

ADDINFO orchestrates this section of the program.

First, it loads in a file that contains ***no procedures*** but does contain the existing alphabetical index stored as a variable.

Next, it initializes the variable "NEWBATCH that will temporarily store the list of names that are being added in this particular session. This list (:NEWBATCH) will be used later by the SORT procedure, and the names in :NEWBATCH will be inserted into :ALPHAINDEX.

The recursive ADDFRIEND runs, and while it runs the subprocedure COLLECT continues to build up the list :NEWBATCH.

When ADDFRIEND ends, a new alphabetical index is created by inserting the names of the newly added friends into the existing alphabetical index. Which line in ADDINFO causes this to happen? _____(q)

The last five lines in ADDINFO are needed to save the new :ALPHAINDEX and inform the user what is happening. Readers wanting to understand this bit of higher order Logo nerdism should consult the reference manual to find out about packaging a variable and saving the variable without saving anything else.

If you type in the new procedures and modify the procedures from last month, you will get an error message the first time you run ADDFRIEND. There is no file called "INDEXES to load. Here's what to do.

Step 1: Type ERNS to erase all names and variables.

Step 2: Type MAKE "ALPHAINDEX [ABRAMS KENNEDY MARTINEZ SMITH]. (You should type in the actual names of the records you have created.)

Step 3: Type PPROP "ALPHAINDEX "VALPKG "INDEXES

Step 4: Type SAVE "INDEXES "INDEXES

The ADDINFO section of the data base project is now in pretty good shape. You could improve this section by adding a routine that displays the information and asks you to confirm that the information is correct before it is stored on the disk. You might want to include directions to specify how to enter the birthday field information with a view to creating a birthday index at some future time. Now let's make some improvements in the USEINFO section of the program.

Making USEINFO More Useful

Look at the revised versions of USEINFO and SETUP. USEINFO has been made a recursive procedure by adding some PRINT messages and a conditional that checks to see how the user responds to the question "DO YOU WISH TO LOOK UP SOMETHING ELSE IN YOUR FRIENDS DATA BASE?" Any answer except "YES," "YUP," "YEAH" or "Y" will cause USEINFO to CLOSE all files on the disk that might have been opened, ERASE all variable names and END.

```
TO USEINFO
TEXTSCREEN CLEARTEXT
SETUP
PRINT [ THERE ARE 3 WAYS YOU CAN USE YOUR ]
PRINT [ FRIENDS DATA BASE ]
PRINT []
PRINT [ 1 FIND A SPECIFIC PIECE OF INFOR- ]
PRINT [   MATION ABOUT A SPECIFIC FRIEND. ]
PRINT [   FOR EXAMPLE . . . WHEN IS MARY ]
PRINT [   SMITH'S BIRTHDAY? ]
PRINT []
PRINT [ 2 LIST A SPECIFIC PIECE OF INFOR- ]
PRINT [   MATION ABOUT EACH FRIEND. ]
PRINT [   FOR EXAMPLE . . . LIST THE FAVORITE ]
PRINT [   FOOD OF EACH FRIEND. ]
PRINT []
PRINT [ 3 SEARCH FOR INFORMATION. ]
PRINT [   FOR EXAMPLE . . . DO ANY OF MY ]
PRINT [   FRIENDS LIKE PIZZA? ]
PRINT []
PRINT [ TYPE THE NUMBER FOR THE WAY YOU WOULD ]
PRINT [ LIKE TO USE YOUR DATA BASE. ]
MAKE "CHOICE READWORD
IF :CHOICE="1 [ GETINFO ]
IF :CHOICE="2 [ FINDALL ]
IF :CHOICE="3 [ SEARCH ]
PRINT [] PRINT []
PRINT [ DO YOU WISH TO LOOK UP SOMETHING ELSE -
  IN YOUR FRIENDS ]
PRINT [ DATA BASE? TYPE  Y  FOR "YES" OR  N  FOR -
  "NO" ]
PRINT []
IF FIRST READLIST="Y [ USEINFO ]
CLOSEALL
ERNS
END

TO SETUP
ERNS
LOAD "INDEXES
MAKE "FIRSTNAME 0
MAKE "LASTNAME 9
MAKE "PHONE 25
MAKE "BIRTHDAY 36
MAKE "PETS 50
MAKE "FOOD 100
MAKE "BUFFER []
MAKE "STORAGE []
MAKE "SEARCHSTORAGE []
END
```

SETUP now contains the line LOAD "INDEXES. This will ensure that the user is always using the most up-to-date indexes to control the processing of the records. Now examine the GETINFO and SHOW.FILE procedures.

```
TO GETINFO
CT
PRINT [ YOU CAN FIND OUT ABOUT THE FOLLOWING -
  FRIENDS ]
PRINT []
SHOWINDEX :ALPHAINDEX
PRINT []
PRINT [ WHICH FRIEND DO YOU WANT TO FIND OUT -
  ABOUT? ]
MAKE "RECORDNAME READWORD
PRINT []
PRINT [ YOU CAN FIND OUT YOUR FRIEND'S: ]
PRINT [    FIRST NAME ]
PRINT [    LAST NAME ]
PRINT [    PHONE NUMBER ]
PRINT [    BIRTHDAY ]
PRINT [    FAVORITE PETS ]
PRINT [    FAVORITE FOOD ]
PRINT [    ALL THE ABOVE ]
PRINT []
PRINT [ TYPE IN ONE OF THE FOLLOWING WORDS: ]
```

```
  PRINT []
✓ PRINT [ FIRSTNAME LASTNAME PHONE BIRTHDAY PETS →
    FOOD ALL ]
  MAKE "FIELDNAME READWORD
✓ TEST :FIELDNAME = "ALL
✓ IFTRUE [ SHOW.FILE :RECORDNAME ]
✓ IFTRUE [ STOP ]
  MAKE "FIELDPOSITION THING :FIELDNAME
  CT
  TYPE :RECORDNAME
  REPEAT 8 [ TYPE CHAR 32 ]
  PRINT RETRIEVE :RECORDNAME :FIELDPOSITION
  END

  TO SHOW.FILE :RNAME
  CLEARTEXT
  OPEN :RNAME
  SETREAD :RNAME
  SETREADPOS 0
  SETWRITE "CON
  PRINT SENTENCE [        FIRST NAME: ] READLIST
  SETREADPOS 9
  PRINT SENTENCE [         LAST NAME: ] READLIST
  SETREADPOS 25
  PRINT SENTENCE [     PHONE NUMBER: ] READLIST
  SETREADPOS 36
  PRINT SENTENCE [          BIRTHDAY: ] READLIST
  SETREADPOS 50
  PRINT SENTENCE [     FAVORITE PETS: ] READLIST
  SETREADPOS 100
  PRINT SENTENCE [    FAVORITE FOODS: ] READLIST
  SETREAD "CON
  CLOSE :RNAME
  END
```

GETINFO has been revised to allow the user to show the entire file of information for a specific friend. Notice the TEST statement followed by two lines that begin with IFTRUE. What happens if you remove the line IFTRUE [STOP] ?

(r)

The purpose of the SHOW.FILE procedure is to display information from a friend's record in a compulsively neat format. To see the alternative, type POFILE "ABRAMS.

Review Part 1 of this series to gain a better understanding of SHOW.FILE.

A file is opened. (For example, "ABRAMS.)

The reader is directed to read the disk instead of the keyboard.

The *reader*'s position is set to position 0—the place where the list containing Rebecca Abrams' first name resides on the disk.

The *writer* is directed to the console—an ancient word meaning the keyboard and screen you are using.

A sentence is printed (FIRST NAME: REBECCA). Notice how all the colons are nicely lined up. One way to do this is to use \ to prevent these spaces from being collapsed.

The above process is repeated until all the information has been printed on the screen. Then the *reader* is directed back to the console and the file is closed.

The Search Option

The main addition to the USEINFO section of the program is an option that permits a person to search for the records that contain a certain item in a selected field. The SEARCH routine presented below is rudimentary. It only looks for exact matches. Thus, if Bill Smith typed in "A CAT" for a pet and Mary Jones typed in "CAT," this search routine would not find both records. First, look back at figure 2 to see how the various procedures are related and then examine the procedures and the explanatory comments.

```
  TO SEARCH
  TEXTSCREEN CLEARTEXT
  SEARCH.DIRECTIONS
  PRINT []
  PRINT []
  PRINT [ THE COMPUTER WILL SEARCH FOR RECORDS ]
  PRINT ( SENTENCE [ SHOWING ] :SPECIFIC [ IN THE ] →
    FIELDNAME [ CATEGORY ] )

  PRINT []
  PRINT [ PRESS A KEY TO CONTINUE ]
  MAKE "CONTINUE RC
  SEEK.ALPHAINDEX :FIELDPOSITION :SPECIFIC
  CLEARTEXT
  PRESENT
  INSPECT RECORDS
  END


  TO SEARCH.DIRECTIONS
  PRINT [ YOU CAN LOOK THROUGH YOUR FRIENDS ]
  PRINT [ DATA BASE AND SEARCH FOR INFORMATION ]
  PRINT [ IN THE FOLLOWING CATEGORIES: ]
  PRINT []
  PRINT [ FIRSTNAME LASTNAME PHONE BIRTHDAY PETS →
    FOOD ]
  PRINT []
  PRINT [ TYPE IN ONE OF THE CATEGORY NAMES ]
  PRINT [ AS IT APPEARS ABOVE. ]
  PRINT []
  MAKE "FIELDNAME READWORD
  MAKE "FIELDPOSITION THING :FIELDNAME
  CLEARTEXT
  PRINT [ THE NEXT STEP IS TO DECIDE WHAT ]
  PRINT [ SPECIFIC INFORMATION TO HAVE THE ]
  PRINT [ COMPUTER LOOK FOR. ]
  PRINT []
  PRINT [ FOR EXAMPLE . . . IF YOUR ]
  PRINT [ CATEGORY WAS 'PETS,' MAYBE YOU WANT ]
  PRINT [ TO FIND OUT IF THE WORD 'GERBIL' ]
  PRINT [ APPEARS ANYWHERE. ]
  PRINT []
  PRINT [ TYPE IN THE WORD OR PHRASE YOU WANT ]
  PRINT [ THE COMPUTER TO SEARCH FOR IN THE ]
  PRINT ( SENTENCE :FIELDNAME [ CATEGORY ] )
  PRINT []
  MAKE "SPECIFIC READLIST
  END
```

SEARCH uses four subprocedures to do its work. SEARCH.DIRECTIONS explains to the user what to type and records information in three variables that will be used subsequently.

MAKE "FIELDNAME READWORD stores the name of the field that the user is going to search within.

MAKE "FIELDPOSITION THING :FIELDNAME translates the field name just entered into a number. The correspondence between a particular field name and a number

is set up in the SETUP procedure. If the person types in PHONE for field name, what number will :FIELDPOSITION be? _____(s)

MAKE "SPECIFIC READLIST stores the item that the person is searching for. In the food field, you might expect a user to search for persons who like PIZZA, for example.

```
?PO [ SEEK MATCHP ]
TO SEEK :INDEX :FIELDPOSITION :SPECIFIC
IF :INDEX=[] [ STOP ]
OPEN FIRST :INDEX
SETREAD FIRST :INDEX
SETREADPOS :FIELDPOSITION
MAKE "FIELD READLIST
TEST MATCHP :SPECIFIC :FIELD
IFTRUE [ MAKE "SEARCHSTORAGE LPUT FIRST :INDEX -
  SEARCHSTORAGE ]
CLOSE FIRST :INDEX
SETREAD "CON
SEEK BUTFIRST :INDEX :FIELDPOSITION :SPECIFIC
END

TO MATCHP :INPUT1 :INPUT2
TEST EQUALP :INPUT1 :INPUT2
IFTRUE [ OUTPUT "TRUE ]
TEST MEMBERP :INPUT1 :INPUT2
IFTRUE [ OUTPUT "TRUE]
OUTPUT "FALSE
END
```

The SEEK procedure opens each record listed in the ALPHAINDEX, positions the reader at the correct field position, and then runs a TEST to see whether the information being read matches the :SPECIFIC being searched for. If the information *does* match, then the record name is stored. What variable stores this record name and where was that variable initialized? _____(t)

SEEK is recursive and processes each record listed in the ALPHAINDEX before ending.

Now examine the last two subprocedures (and their attendants) called by SEARCH.

```
TO PRESENT
TEST :SEARCHSTORAGE=[]
IFTRUE [ PRINT [ SENTENCE [ THERE ARE NO RECORDS -
  WITH ] :SPECIFIC [ IN THE ] :FIELDNAME -
  [ CATEGORY ] ) ]
IFTRUE [ STOP ]
IFFALSE [ PRINT [ SENTENCE [ THE FOLLOWING -
  RECORDS CONTAIN ] :SPECIFIC [ IN THE ] :FIELDNAME -
  [ CATEGORY ] ) ]
IFFALSE [ PRINTLIST :SEARCHSTORAGE ]
END

TO PRINTLIST :LIST
IF :LIST=[] [ STOP ]
PRINT FIRST :LIST
PRINT []
PRINTLIST BUTFIRST :LIST
END
```

```
TO INSPECT.RECORDS
TEST :SEARCHSTORAGE=[]
IFTRUE [ STOP ]
PRINT [] PRINT []
PRINT [ TYPE IN THE NAMES OF THE RECORDS YOU -
  WISH TO INSPECT. ]
PRINT []
PRINT [ JUST PRESS ENTER IF YOU DON'T WANT TO SEE -
  ANY OF THE ]
PRINT [ ABOVE RECORDS. ]
PRINT []
MAKE "RECORDLIST READLIST
SHOW.RECORDS :RECORDLIST
END

TO SHOW.RECORDS :LIST
IF :LIST=[] [ STOP ]
SHOW.FILE FIRST :LIST
PRINT [] PRINT []
PRINT [ PRESS A KEY TO CONTINUE ]
MAKE "CONTINUE RC
SHOW.RECORDS BUTFIRST :LIST
END
```

PRESENT (and its attendant, PRINTLIST) informs the user whether there are any records that contain the information being searched for. Notice the TEST . . . IFTRUE [STOP] construction in PRESENT. Remove this line and then run the SEARCH procedure while looking for an item that you know does not appear in any of the records. What happens?

The final subprocedure in the SEARCH routine is INSPECT.RECORDS. It records the names of the records the user may want to inspect. What line records this information? _____(u)

Each record is displayed by SHOW.RECORDS. Notice how this recursive procedure displays the information and then waits for the user to press a key to continue.

Concluding Remarks

There is much left to improve in this data base project. For example:

- More complicated searches should be possible.
- A section needs to be created that permits the updating of information without requiring that the entire record be retyped.
- A section needs to be created that allows reports to be printed and standard report formats stored for repeated use.
- Confirmation prompts could be added so that a user must give pause and reflect before sending the computer on its merry way to do "what I said but not what I meant."

Finally, this data base now requires too much careful typing. The choices should be made either by selecting numbers from a menu or by moving the cursor to a certain item and pressing a certain key.

We hope that readers have enjoyed moving beyond turtle graphics with Logo. Our hope is to lay to rest the claim that there is nothing to Logo except graphics. **END■**

[Tim Riordon, Associate Director of Curriculum and Training, National Computer Training Institute, 39510 Paseo Padre Parkway, Fremont, CA 94556.]

BEST COPY AVAILABLE

Animation with M.I.T. Logo

by
Nancy Parks Sopp

A bird circles once over an island. A clover plant sprouts. Soon clover dots the island. A turtle meanders across a hillock. The bird returns and circles again.

Andrew, a seventh grader, created this dynamic scene for his social studies class. He made the island with a Koala Pad and added life with M.I.T. Logo.

M.I.T. Logo has a limited capacity for animation.[1] You can, however, make the Logo turtle into a spaceship, a butterfly—whatever you like. You can then make the shapes move. By changing back and forth from one shape to another similar shape, you create the illusion that the shape itself is in motion.

The SHAPE.EDIT file on the utilities disk contains the procedures that allow you to create a new shape for the turtle by plotting points on the hi-resolution screen. Instead of drawing with the turtle, you redesign its body.

Andrew learned many new programming techniques through this project. The intrigue of working on animation brought him into the computer lab for extra Logo time every day.

First Andrew created the island with a Koala Pad.[2] Andrew then created two separate bird shapes—one with outstretched wings and one with curved wings (see figure 1). Andrew's procedure tells the (invisible) turtle to change back and forth from shape A to shape B many times while it moves forward in an arc. This creates the illusion of the bird's wings flapping while it moves across the screen (see figure 2).

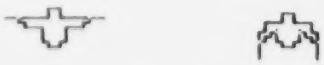


Figure 1

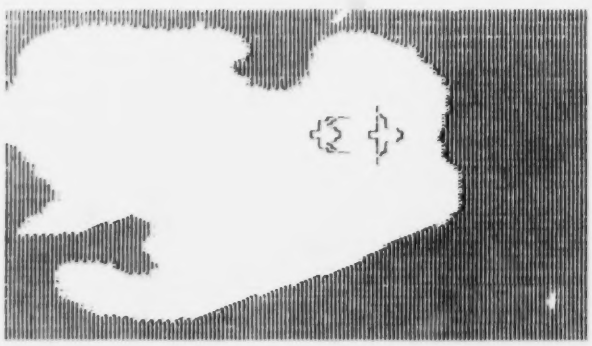


Figure 2

The turtle disappears at the edge of the screen and reappears, this time as a clover flower. Andrew used the .STAMP procedure from the SHAPE.EDIT file to make the turtle leave its clover image all over the island (see figure 3).

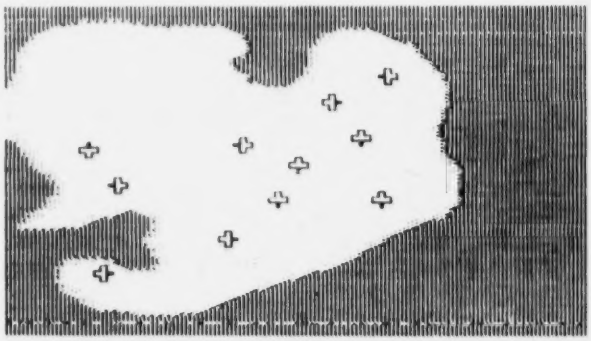


Figure 3

The turtle disappears once more and reappears as a tortoise. Andrew made the tortoise crawl along the *X* axis with these commands: REPEAT 20[SETX XCOR + 1 WAIT 5] (see figure 4).

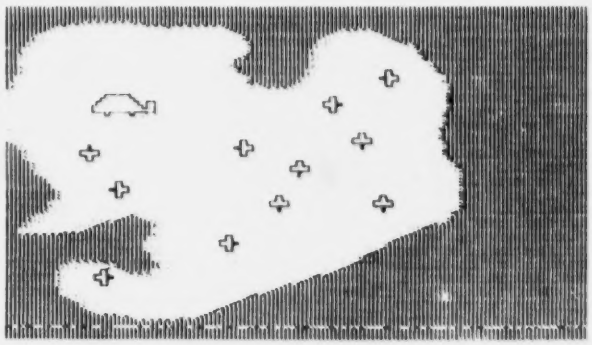


Figure 4

He then stopped the tortoise's movement and stamped the shape on the screen. The turtle took on the bird shapes—switching from one to the other repeatedly—and circled the screen again.[3]

Creating

Sparse instructions for changing the turtle's shape appear in the technical manual. I'll elaborate on the steps here.

Let's create two shapes—ROCKET and TORTOISE—and save them in a file called TURTLES.

First, insert the utilities disk into drive 1 and type

```
READ "SHAPE.EDIT
```

This reads into memory and displays on the screen the assembly language interface procedures you'll need to create your shape. Type

```
SETUP
```

and you're ready to create your first shapes.

Type

```
MAKESHAPE "ROCKET
```

The screen is now blank, except for one line of print at the bottom:

```
DEFINING SHAPE ROCKET
```

Use these keys to plot points:

CTRL-P	Move up
CTRL-N	Move down
Arrow keys	Move right and left
ESC	Delete one or two commands
1 . . . 9	Set the shape's size
U	Penup
D	Pendown

Caution: Make your shape small, or you may run out of workspace before you've completed your second shape. You can increase the shape's scale while you're creating it: Press a number between 1 and 9. (The default size is 1.)

When your shape is complete, press CTRL-C to define your shape. CTRL-G aborts the creation process if you want to start over again.

Create your second shape. Type

```
MAKESHAPE "TORTOISE
```

Use the keys as in the table above to plot the points. Then define the shape by pressing CTRL-C. Practice swapping the shapes:

```
SETSHAPE :ROCKET
SETSHAPE :TORTOISE
```

Restore the turtle's original triangular shape with this command:

```
SETSHAPE 0
```

(You may need to type SIZE 1.)

Comments on Shape Creation

After you've defined your shape, you can use the SIZE command to set the scale: SIZE 1 through SIZE 255 (see figure 5).

Figure 5

When you retrace a line with the pen down, all but a tiny dot is erased. Put the pen up if you wish to retrace a path and preserve the line.

ESC deletes the previous byte in the shape table. A byte represents one or two commands. You can press ESC repeatedly to delete many segments.

The turtleshape's heading is 0. Its pen is positioned at the point from which you begin plotting. You might consider creating an animal shape standing on its tail, so that it will move nose first when you type FD 50. Or plan to move shapes by using X and Y coordinate commands (e.g., REPEAT 10[SETX XCOR + 5]).

Once you've defined a shape with CTRL-C, you can't edit it. If you're dissatisfied with your shape, type

```
MAKESHAPE "ROCKET
```

to recreate it.

Twice I've been halted in my creating because the command keys inexplicably stopped working. I finally pressed CTRL-G and recreated the shape.

You can create several shapes before you save them. In fact, if you intend to use several shapes in one procedure, as Andrew did, you must create them at the same time and save them in one file. If you read successive shape files into memory, the shapes in your first file will look distorted.

I haven't been able to create more than seven shapes at once: Logo kicks me out of the shape editor with the message, TOO MANY SHAPES! If your shapes are large, the TOO MANY SHAPES! message will appear sooner. If this happens, you can recreate your last shape or stop and save your shapes to disk (see the next section). Saving your shapes to disk will clear the workspace, thus enabling you to create more shapes.

Practice moving your shapes around on the screen, using regular graphics commands (e.g., FD 50).

Saving Shapes

To save your procedures, type

```
SAVESHAPES "TURTLES
```

Press ⟨return⟩. The following instructions will appear:

```
PLACE THE FILE DISK IN THE DISK DRIVE AND PRESS
RETURN
```

Do so, using an initialized disk. The SAVESHAPES superprocedure creates two files—one named TURTLES.AUX.LOGO and one named TURTLES.SHAPES. The TURTLES.SHAPES file contains the names of the shapes and the procedures you'll need when you retrieve the shapes.

After your shapes have been saved, the following message appears:

```
PLACE THE DISK CONTAINING THE SHAPE EDITOR IN THE
DISK DRIVE AND PRESS RETURN
```

The SAVESHAPES procedure erased the shape-creating procedures from memory, so as not to unnecessarily clutter your shapes files. If you wish to create more shapes, press ⟨return⟩. The shape editor will be read back into memory. If you don't wish to continue, press CTRL-G.

Retrieving Shapes

Let's practice now how you'll retrieve your shapes from your disk at your next Logo session. First, type GOODBYE to clear the workspace. Place your files disk in the disk drive. Type

```
READ "TURTLES.AUX
```

The procedures INITSHAPES, SIZE, SETSHAPE and .SHAPE are read into memory. Type

```
INITSHAPES
```

The INITSHAPES procedure loads the shape tables into memory.

Type

```
SETSHAPE :ROCKET
```

Type

```
SETSHAPE :TORTOISE
```

If you followed the instructions carefully, your shapes should have appeared.

Comments

The shape file that you retrieve has ".AUX" behind the name. Be sure to type that when you retrieve your file.

INITSHAPES loads your shape table, TURTLES.SHAPES. When you saved your shapes, the INITSHAPES procedure was created. Edit it to see what it does.

If you've forgotten the name you gave your shape, type ED ALL. You'll find your shape name among the MAKE statements at the end of the edit buffer.

Your shape will move in every direction, but can only face four directions: HEADINGs 0, 90, 180, 270. You'll see what I mean if you give your shape instructions to draw a circle: REPEAT 180 [FD 1 RT 2].

Use SIZE to change your shape's scale. Students like to write procedures that do nothing but change the scale:

```
TO FREAKOUT
SETSHAPE :TORTOISE
REPEAT 10 [ SIZE 1 SIZE 2 SIZE 3 SIZE 4 SIZE 5 ]
```

The .SHAPE 1 Command

This lets you stamp your shape onto the screen, as Andrew did with his clover. You must stamp your shape if you plan to save and print out its image in a picture. The following procedure will stamp your shapes in four locations on the screen.

```
TO STAMP
PU
SIZE 2
SETHEADING 0
SETSHAPE :ROCKET
SETXY (-50) 50
.SHAPE 1
SETSHAPE :ROCKET
SETXY 50 (-50)
.SHAPE 1
SETSHAPE :TORTOISE
SETXY 50 50
.SHAPE 1
SETSHAPE :TORTOISE
SETXY (-50) (-50)
.SHAPE 1
END
```

Comments

I haven't explained how ".SHAPE 1" stamps the shape on the screen because I don't understand it. The technical manual describes it briefly.

You must tell the turtle what shape to take on after you've stamped it—either SETSHAPE :ROCKET for the shape or SETSHAPE 0 for the triangular turtle. If the turtle is still at the position where you stamped it, the shape will disappear. When you move the turtle, however, both the stamped shape and the dynamic one will appear.

Moving Your Shape

Make your shape move slowly by having it REPEAT small steps. Here are four examples. You can make your rocket blast off with the first two.

```
REPEAT 20 [ FD 5 ]
REPEAT 20 [ SETY YCOR+5 ]
```

Make your tortoise draw a circle with the following command.

```
REPEAT 180 [ FD 2 RT 2 ]
```

Use the next command to make your tortoise scuttle across the screen.

```
REPEAT 20 [ SETX XCOR+5 ]
```

Animation

Andrew's flying procedure alternates bird shapes while the turtle moves.

```
TO FLY
PU
SETXY 50 (-40)
SETH 270
SIZE 1
REPEAT 18[ SETSHAPE :BIRDA FD 12 RT 10 SETSHAPE
  :BIRDB FD 12 RT 10 ]
REPEAT 7[ SETSHAPE :BIRDA FD 12 SETSHAPE :BIRDB
  FD 12 ]
SETSHAPE :BIRDA FD 12
HT
END
```

I used two shapes to make a stick figure walk across the screen (see figure 6).

Figure 6

```
TO WALK
SIZE 2
REPEAT 10 [ MANWALK MAN2WALK ]
END

TO MANWALK
PU HT SETSHAPE :MAN
ST SETX XCOR+5
END

TO MAN2WALK
HT SETSHAPE :MAN2 ST
SETX XCOR+5
END
```

Printing

Stamp your shape with .SHAPE 1 before you save a picture. If you don't the shape will disappear, just as the normal turtle does when you use SAVEPICT. (You needn't bother stamping if you use a special graphics interface card to print your pictures directly from Logo, as these cards will always print the turtle image, stamped or not.) **END**■

Nancy Parks Sopp, P.O. Box 81706, Fairbanks, AK 99708.]

NOTES

1. More extensive animation is made possible through the use of an additional card—a "sprite" board—with the M.I.T. versions and with the new Apple II. Logo for the Commodore, Coleco Adam and Texas Instruments computers has sprite capabilities.

2. Andrew created his island picture with a Koala Pad and the software that accompanies it. He saved it, booted up Logo and used the following commands to retrieve it:

```
DRAW
DOS [ BLOAD PICTR.ISLAND,A$2000 ]
```

"DOS" is Logo's disk-operating system command.

Koala pictures appear on page two of the hi-res screen. Logo uses page one. To have the turtle show on the Koala picture, Andrew saved it (SAVEPICT "ISLAND) and then retrieved it (READPICT "ISLAND). He didn't need the DOS command thereafter.

3. Andrew brought all the elements of his island scene together in a superprocedure.

```
TO ISLAND
READ "ANDREW'S.AUX
INITSHAPES
READPICT "ISLAND
FLY
PLANT
CREEP
FLY
END
```

The ANDREW'S.AUX file contains all three shapes. The individual shapes are set within the FLY, PLANT and CREEP procedures, respectively.

Kathleen Martin • Tim Riordon

Tim Barclay—Contributor

Editor's note: Tim Barclay has responded to our request to hear from people who use Logo for math explorations. The personal note which accompanies his submission reveals his sensitivity to how Logo should be taught. He writes:

"I guess I should say something about my intent and bias as I write these ideas. I am bothered by articles which print out Logo programs which are elegant but unreadable for anyone but Logo experts. The Logo philosophy of providing a language in which students can do their own meaningful explorations is central to our use of Logo in schools. So I am not so much interested in showing solutions here as in suggesting topics and problems where kids at the middle school and junior high level can get involved doing math in Logo—or Logo in math. I have written Logo programs for all these problems but am only including a listing for YOUSQRT here."
Tim is working with MIT Logo.

For a long time I always began Logo by entering DRAW as soon as the Logo disk had booted. But there is another way to work with Logo. You can type CLEARTEXT instead and have a full text screen on which you can do mathematics, among other things. At the simplest level you can use Logo like a hand calculator. For instance, enter PRINT 45 + 34 or enter PRINT 255*11. (The arithmetic operators in Logo are + − * and / respectively.)

You can investigate questions of order of operations by entering more complex computations. Try PRINT 24−8*3 and then compare this with PRINT 24−(8*3).

Here are some questions which require thinking about order of operations and parentheses. In each case enter the statement two ways, first without any parentheses and then with parentheses. Of course, if the answer is the same both ways, the parentheses may have been unnecessary.

a) For a 4 by 4 square, what is the area divided by the perimeter?

b) How much change should you get if you pay $5 for three ice cream cones which cost 80¢ each?

c) What is the sum of 1/2+1/3+1/4+1/5?

d) What is the sum of one over two squared (1/(2*2), plus one over three squared plus . . . ?

e) What is the average of your four favorite numbers?

None of these exercises involve much programming in Logo, but they do get you familiar with working with numbers in Logo (and without Logo, too).

Here is a programming problem which involves these arithmetic operations.

In nuclear reactions, an important concept is critical mass. This refers to the mass of radioactive material, uranium or plutonium, necessary for the fission reaction to become self-sustaining. Two opposing factors are involved: the generation of neutrons which can then split more atoms and the leakage of neutrons away at the surfaces of the radioactive material. The generation of neutrons is a function of the volume of radioactive material, and the leakage is a function of the surface area of the material. To see what happens to these two factors as the amount of material gets greater, assume that the uranium is in the shape of a cube. Write a program which prints out the volume, the surface area, and the ratio of volume to surface area for bigger and bigger cubes. Do you see how this leads to a critical mass?

Calculators typically have a square root key on them. Teachers and textbooks have often presented a horrible algorithm for taking square roots—horrible because both as a student and later as a teacher, I could never remember it. An alternative approach is successive approximations. The following program, YOUSQRT, provides a format for finding the square root of a number using this approach. The point of this program is to give students a format for thinking about how to make successively better guesses. It enables you to quickly find a square root to whatever degree of accuracy is desired. The problem is to then go on and write a Logo program which instructs the computer on how to find the square root of any number this way.

```
TO YOUSQRT
  CLEARTEXT
  PRINT [ THIS IS A PROGRAM FOR YOU TO
    FIND ] PRINT [ ]
  PRINT [ THE SQUARE ROOT OF A NUMBER ]
  PRINT [ ] PRINT1 [ ENTER YOUR NUMBER
    . . . ]
```

```
  MAKE "YOURNUM FIRST REQUEST
  CLEARTEXT
  SQRTGUESS
  CLEARTEXT CURSOR 3 8
  PRINT ( SE [ SO YOU THINK THE SQUARE
    ROOT OF ] :YOURNUM "IS :GUESS [ . . .
    LET'S SEE. ] )
  PRINT [ ] PRINT [ ]
  PRINT ( SE :GUESS "TIMES :GUESS "=
    :GUESS * :GUESS )
  PRINT [ ] PRINT ( SE [ DOES THAT EQUAL ]
    :YOURNUM "? )
END

TO SQRTGUESS
  PRINT [ ]
  PRINT1 [ WHAT IS YOUR GUESS? . . . ]
  MAKE "GUESS FIRST REQUEST
  PRINT [ ] PRINT ( SE :YOURNUM [ DIVIDED
    BY ] :GUESS "= :YOURNUM / :GUESS )
  PRINT1 [ . . . . GUESS AGAIN? (Y/N) . . . . ]
  MAKE "ANS RC
  IF :ANS = "N STOP
  SQRTGUESS
END
```

The sequence of activities here begins with a program for experimenting with various ways of making successively better guesses. Here the microcomputer is a manipulative tool. The next step is to think about how you are thinking, and then finally to teach the computer your algorithm. There is no single way through this set of activities and no single right answer, but there is opportunity for everyone to have success at some level. This is one of the gifts Logo and the microcomputer offers to a class.

Here are some questions which are suggested by this activity.

- How many guesses are needed to get within 0.1 of the square root of the number? . . . within 0.01? . . . within 0.00001?
- Does the answer to the above question depend on the size of the original number?
- Does the answer to the first question depend upon making a close first guess? Test this for finding the square root of 300; compare first guesses of 5, of 15, of 17, and of 100.
- When you put in large numbers, say bigger than 1000, how do you make a good first guess?
- Did you try finding the square root of numbers between 0 and 1?
- Did you try finding the square root of negative numbers?

Finding factors of numbers and then investigating properties of these factors offers another rich area of math and Logo opportunities. The Logo primitive REMAINDER can be used here to test whether or not a number is a factor of another number.

```
TEST REMAINDER :NUMBER :DIVISOR = 0
```

Your first problem is to write a program that will find all the factors of any number. Once you have written this you might look at the number of factors for different numbers. Try filling in a table like the one below for the numbers from 1 to 100. Use Logo to find the factors, and use pencil and paper to make and fill in the table.

Factors Table: Enter in each column the numbers which have that number of factors.

No. of Factors	1	2	3	4	5	6	7	8	9	10	11	...

Here are some questions suggested by this Table.

- Which numbers are prime?
- Why are there more numbers with an even number of factors than with an odd number of factors?
- What is the smallest number with more than 10 factors?

A second investigation follows from this work with factors. The factors of 6 are 1, 2, 3 and 6. The proper factors of a number are all the factors except the number itself; in this case, 1, 2 and 3, but not 6. If you add the proper factors of 6 together, you get 6. That makes 6 a **perfect number**, the name for numbers where the sum of the proper factors equals the number itself. The other possibilities are that a number is either **deficient** (sum of the proper factors is less than the number itself) or **abundant** (sum of the proper factors is greater than the number itself). You can use your factor program as a procedure in a program which looks to see whether a number is **perfect, deficient,** or **abundant.**

- There are only four perfect numbers less than 9000. Can you find them?
- What is the frequency of each of these types of numbers between 1 and 100? . . . between 101 and 200? . . . between 1001 and 1100 ? etc.
- What kinds of numbers are the most deficient?
- Why aren't any odd numbers abundant?

Here we have started with a common topic, factors of a number, and branched out into a rich arena of possible math investigations. The microcomputer is a tool which makes this feasible.

If any reader tries these ideas with students, I would like to hear about it. END■

[Tim Barclay, TERC, 8 Eliot St., Cambridge, MA 02138.]

[Tim Riordon, 3375 University, Eugene, OR 97405. Kathleen Martin, Dept. of Education, Univ. of Dallas, Dallas, TX 75061.]

THE LOGO CENTER

Kathleen Martin • Tim Riordon

Polyspirals

If you have never experienced the fascination (and sometimes the frustration) of polyspirals, you simply must try your hand at it. Polyspirals are polygons that change in size as they are drawn. They may either increase or decrease. The ones that we will be working with increase in size. Let's take a look at a few.

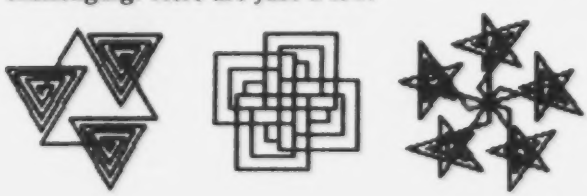
Which of the polyspirals above is described by the following procedure?

```
TO POLYSPIRAL :DISTANCE
IF :DISTANCE > 50 [STOP]
FD :DISTANCE
RT 144
POLYSPIRAL :DISTANCE + 4
END
```

That's right. The procedure fits the star. You can tell because the turning angle for a star is 144 degrees. If we wanted the procedure to draw the square spiral, we would simply change the angle to 90 degrees instead of 144 degrees. It's easy enough to make the procedure more flexible so that it can spiral any polygon of our choice. We can just set up the angle as a variable. The procedure would then read as follows:

```
TO POLYSPIRAL :DISTANCE :ANGLE
IF :DISTANCE > 50 [STOP]
FD :DISTANCE
RT :ANGLE
POLYSPIRAL :DISTANCE + 4 :ANGLE
END
```

Now let's see if we can take our POLYSPIRAL procedure and use it to draw some designs that are both aesthetically pleasing and mathematically challenging. Here are just a few.

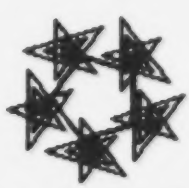


Although they seem elaborate, the designs are generated by a simple-enough procedure. We need to introduce a third variable to take care of the number of times that we want POLYSPIRAL drawn. Then it's only a matter of drawing POLYSPIRAL, moving to a new position and redrawing it.

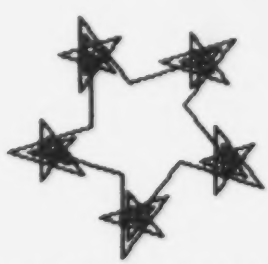
```
TO POLYSPIRALS :NUMBER :DISTANCE :ANGLE
REPEAT :NUMBER [POLYSPIRAL :DISTANCE
  :ANGLE RT :ANGLE FD 40]
END
```

The same turning angle is used in POLYSPIRALS as in POLYSPIRAL. Do you know why? Later you can try using two different turning angles, but for now let's stay with what we have.

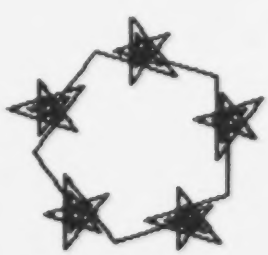
Let's spin a few POLYSPIRALS keeping the NUMBER and ANGLE constant and changing only the DISTANCE. If we use a NUMBER of 5 and an ANGLE of 144, we guarantee that we will always have 5 stars (or almost stars). Under each of the designs below, the DISTANCE variable is the changing number associated with the procedure name.



POLYSPIRALS 5 1 144

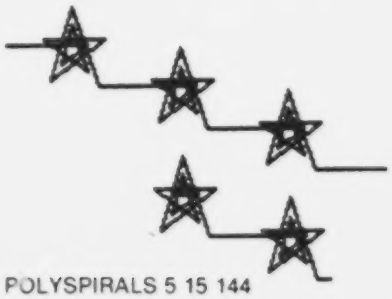


POLYSPIRALS 5 3 144

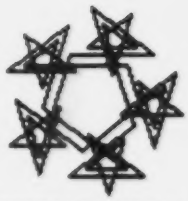


POLYSPIRALS 5 7 144

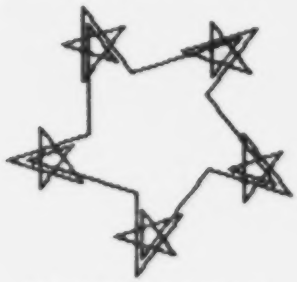
POLYSPIRALS 5 11 144



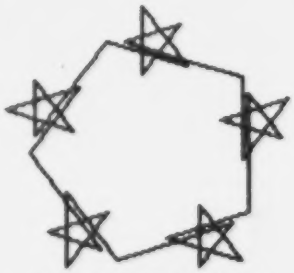
POLYSPIRALS 5 15 144



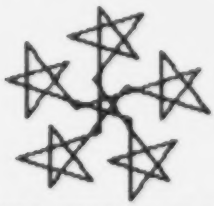
POLYSPIRALS 5 19 144



POLYSPIRALS 5 23 144

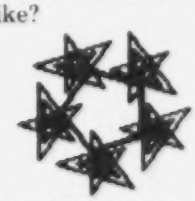


POLYSPIRALS 5 27 144



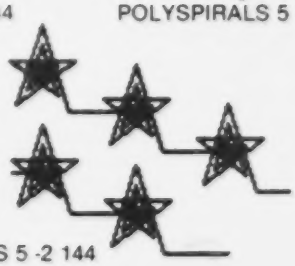
POLYSPIRALS 5 31 144

Do you see a pattern in the DISTANCE variable? Except for POLYSPIRALS 1 and POLYSPIRALS 3, there is a numeric difference of 4 between any two successive POLYSPIRALS. Is that coincidental? And what about the first two? Let's examine the last question. POLYSPIRALS for 0, -1 and -2 are given below. Notice that 0 and -1 are like POLYSPIRALS 3 above. Which one is POLYSPIRALS -2 like?



POLYSPIRALS 5 0 144

POLYSPIRALS 5 -1 144

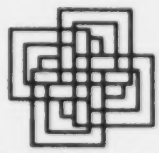


POLYSPIRALS 5 -2 144

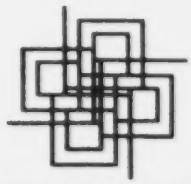
All kinds of wonderful questions are raised by careful examination of the POLYSPIRALS group. Why does the design change after every four of the POLYSPIRALS? How many basic kinds of designs are there? Doesn't POLYSPIRALS 3 resemble POLYSPIRALS 23? And look at the similarity between POLYSPIRALS 15 and POLYSPIRALS 35. Does the same thing happen with squares as happens with stars?

Let's examine that last question and see if it can help us in addressing the other two. To draw square spirals, all we have to do is to use an ANGLE of 90 and NUMBER of 4 as our constants. The DISTANCE variable is again all that we will change. Below are some square spiral designs. What do you notice this time?

POLYSPIRALS 4 -1 90



POLYSPIRALS 4 3 90



POLYSPIRALS 4 7 90

POLYSPIRALS 4 11 90

The same thing seems to be happening. A new pattern emerges after every fourth design. How many different kinds of patterns are there all together? Is this the same as for the star? (Think about the number of different headings involved in drawing a star.)

Try changing the increment in the POLYSPIRAL procedure. For all of the designs generated thus far, the increment used has been 4. Change the increment to 3 as follows:

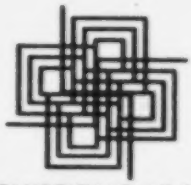
```
TO POLYSPIRAL :DISTANCE :ANGLE
IF :DISTANCE > 50 [STOP]
FD :DISTANCE
RT :ANGLE
POLYSPIRAL :DISTANCE + 3 :ANGLE
END
```

What difference does this make in the pattern of the designs below? There still seems to be only four kinds of patterns, but how often are they repeating? Why the change? If the conditional statement of the procedure used a distance of 30 rather than 50, how would the patterns be affected?

POLYSPIRALS 40 0 90

POLYSPIRALS 4 3 90

POLYSPIRALS 4 6 90



POLYSPIRALS 4 9 90

By now you are either terribly frustrated or wonderfully excited. If you are among the teeth-grinding group, go back to the POLYSPIRALS procedure and just play with the ANGLE and NUMBER variables. You will find that you can generate all kinds of delightful designs. The sheer delight of the many possibilities is simply another dimension of learning.

For those of you who want to explore further the questions previously raised, it should help to study the positions of the turtle in the POLYSPIRAL procedure. When an ANGLE of 90 degrees is used, the turtle has four possible headings. Notice which DISTANCEs result in the same heading. Also try examining star POLYSPIRALS and triangular POLYSPIRALS. If you come up with some answers, that's great. However, answers are not necessary. There is much to be learned from simply becoming aware of patterns and noting relationships.

POLYSPIRAL procedures offer students lots of challenging questions to pursue. It is particularly helpful if a printer is available for dumping the graphics. Designs can be labeled, cut out and arranged in groups that facilitate the observation of relationships. If a printer is not available, the students should sketch the design. Tissue paper can be held against the screen and the designs traced for greater accuracy.

* * *

Next month's column will deal with the topic of Logo and numbers. Many middle school teachers weave BASIC into their math classrooms. What are the possibilities and limitations for using Logo in this context? END■

[Tim Riordon, 3375 University, Eugene, OR 97405. Kathleen Martin, Dept. of Education, Univ. of Dallas, Dallas, TX 75061.]

Helping Students with Recursion: Teaching Strategies

[Permission to reprint is granted provided full credit is given the author and The Computing Teacher.*]*

This paper has been inspired by the work of three other people. In a paper presented at the Fifth Annual Conference of the Cognitive Science Society, Kurland and Pea presented some disturbing findings about children working with the concept of recursion in Logo; namely, that students with a year of Logo programming are not understanding recursion. Kurland and Pea believe that their student subjects were misguided because they had constructed an incorrect "looping model" for recursion. A looping model works successfully in the context of tail recursion but fails when students encounter embedded recursion. These findings caused me to begin thinking about how to provide students with better mental models for recursion. As I began thinking about this problem, I found myself recalling the ideas that D. R. Hofstadter presented in a public lecture entitled "Recursion in Nature and Art." I wish to acknowledge Douglas Hofstadter as the inspiration and source of many of the ideas presented below.

"If students are to understand the concept of recursion, then that understanding should be grounded in a rich background of non-computer recursive experience."

The first mistake many of us make in presenting the concept of recursion is that we attempt to introduce the idea in a computer context. If students are to understand the concept of recursion, then that understanding should be grounded in a rich background of **non-computer recursive experience.** Students learn a turtle geometry, built upon the idea of defining everything in terms of moving and turning relative to one's present location and heading. Their learning is made possible and facilitated because of their rich background in moving and turning their bodies. I believe that my own past attempts to teach recursion to children were as misguided as it would be to try to teach turtle geometry to someone who lacked a background in seeing and feeling movement and turning in space. The first task in teaching recursion is to help students recall or to provide them with recursive experiences. The activities which follow attempt to do that.

A second mistake made by many of us who have focused upon recursion in a computer context is that our talk, our metaphors and our diagrams too often contribute to students arriving at a "looping" model of recursion. The following lesson plans avoid that mistake.

A third problem is that we teachers are often trapped into teaching a concept which we do not ourselves fully grasp. I believe that this article, in conjunction with Dan Watt's *Learning With Logo* and Hal Abelson's *LOGO for the Apple II,* will help teachers to better understand the powerful computer idea of recursion. So, another purpose of this article is to help teachers acquire a better understanding of this important concept.

These ideas are presented in a lesson plan format in the hope that teachers will actually use the suggested materials and activities.

Part I: Introducing Recursion with Non-Computer Recursive Experiences

TEACHER BACKGROUND

Take a moment to read through Student Activities 1-5, and then return to this background section and consider the following comments:

- The purpose of these activities is to provide students with recursive experiences—experiences that are associated with the word "recursion" itself. By giving students this word, you give them a way to talk about other experiences which they will encounter.
- These activities were selected because they expose students to *objects or processes which contain copies of themselves.* This is the aspect of recursion which we need to reinforce. This is the mental model we want students to employ instead of a "recursion is looping" model. In Logo, recursion involves procedures that in some sense "contain" themselves.

- It is important for students to experience recursion in both an auditory and visual way. Activities 2, 3 and 5 should not be skipped.
- It is important to have students try to actively create recursive productions. Activities 4 and 5 accomplish this.

MATERIALS

- 2 good-sized inexpensive mirrors, 12" x 20". One mirror should have a stand.
- Record player
- 2 tape recorders, one with variable speed control
- Recording of "Windmills of the Mind," a popular song recorded by a number of artists
- Video camera and monitor
- Morton salt container or Land of Lakes butter carton
- Transparencies I-IV (see p. 42)
- Cassette with the recording of "Teacher's Example of Recursion"

In order to create "Teacher's Example of Recursion," you'll need three students, the two tape recorders, 2 cassette tapes (labeled TAPE A and TAPE B), and Transparency III to serve as a script. Assign parts to each student.

1. Turn on the variable speed recorder and record at normal speed the students saying their lines on TAPE A. Encourage students to enunciate clearly and not to rush the tempo. Just skip over the "hole" in the script pattern this first time. Rewind this tape and get it ready to play. It helps if you have a silent microphone switch. Set the playing speed to high (about 25% faster than normal).
2. Now get the second recorder ready to record. This recorder doesn't need to be a variable speed recorder. Turn on the second recorder and have the students begin recording the same lines on TAPE B, but when you come to the "hole" in the script pattern (right after the line "*It sounds like this*"), switch on the first tape recorder. The first copy from TAPE A will then play back at a faster tempo and higher pitch. This is being placed into the "hole" in the script and is now being dubbed onto TAPE B. When this higher-pitched playback is finished, switch it off and have the student continue recording the last line of script onto TAPE B. This is the line that comes after the "hole." Then stop this recorder and rewind it. Play it back to see if you captured the copy of the conversation within the conversation.
3. Rewind the tape you just recorded (TAPE B). Now take that tape and place it in the variable speed recorder. Leave it set to play back at a faster speed. Have the switch ready so you can turn on this recording. Put the TAPE A in the regular tape recorder, erase it and get it set to record.
4. Turn on the regular recorder and begin recording the script on TAPE A. When you come to the "hole" in the script, switch on the variable speed recorder so that TAPE B plays back at a faster speed. Now TAPE B is being dubbed onto TAPE A in the "hole" in the script. This time two higher pitched copies of the script will be heard. When TAPE B is finished playing back, cue your student to say the final line and then cease recording on TAPE A. Rewind TAPE A and play it back. You will now hear a copy within a copy within a copy of the pattern script. Rewind this and place it in the variable speed recorder ready to play back once again.
5. Take TAPE B, erase it, put it in the regular recorder and prepare the recorder to record. Repeat the process until you have a tape that contains 4 or 5 copies within copies.

Don't be discouraged by the poor quality of recording you obtain. The value of this is not in the final product—it's the process that students go through to create the recursive example.

Douglas Hofstadter inspired this idea by playing a tape recording of a recursive ad for his book *Godel, Escher, Bach: an Eternal Golden Braid* (Basic Books, 1979).

ACTIVITIES

Activity 1: Visual Example

Have students take turns putting their heads between two good-sized mirrors which are placed so that they face each other. Allow them to adjust one of the mirrors. Ask them to comment upon what they see (an infinity of reflected images). Encourage students to verbalize the process that produces the infinity of images. (One mirror shows the other mirror that is showing the first mirror showing the other mirror. . . .) Don't over-emphasize the verbalizing aspect; your goal is to produce a right-brain understanding, not a left-brain verbalization. Once all students have experienced this, tell them that this experience has something to do with an idea called recursion. If they ask you, "What's recursion?" tell them that you aren't ready to explain it yet. If they really press you with, "Hey, you're a teacher and you're 'sposed to explain things," tell them that recursion is a process in which something contains a copy of itself. Leave the mirrors in the classroom for students to return to. If your students are older and know something about light and images, you might ask them whether there really are an infinity of images. (There aren't.)

Activity 2: Video Recorder, Microphone and Speaker

Bring a video camera and a monitor into your classroom. Encourage students to point the camera at the video screen. Ask them what they see. Encourage them to turn the camera and to hold it at various angles. Ask them to verbalize what's going on. Again, tell them that this process has something to do with an idea called recursion. There is something that contains a copy of itself. Most video camera setups have microphones and speakers. Turn up the volume on the amplifier and bring the microphone near the speaker. When you get the piercing hum, you can explain that this also has something to do with recursion. Help them understand what caused the hum. (Unfortunately, it's called a "feedback loop." This is unfortunate, because "loop" is the wrong word to use when talking about recursion.)

Activity 3: Musical Example

Play the song, "Windmills of the Mind." Tell students that this has something to do with recursion. Ask them to state the ways in which this song *contains a copy of itself.* Have them sing a round such as "Frere Jacques" or "Row, Row, Row Your Boat." Ask them to discuss whether this is a recursive experience. It isn't important to arrive at a definitive answer to these questions. Sometimes whether something is said to be recursive or not depends upon how the thing is being looked at or considered. Your purpose here is to start students thinking—not to have them *finish* thinking.

Activity 4: Recursive Drawings

Show students the picture on the Morton salt container. It shows a girl pouring a box of salt. The box of salt she is pouring has a picture on it. It is a picture of a girl pouring a box of salt which has a picture on it. The picture is that of a girl pouring . . . etc. Tell students that this has something to do with recursion, because there is something which contains a copy of itself. (Land of Lakes butter also uses a recursive design.)

Next, place Transparency II on top of Transparency I and show it to the class. Ask students whether this picture is recursive. (It is, because it is something that contains a copy of itself.) Remove Transparency II so that only Transparency I is projected. Do your students see that Transparency I is a non-recursive picture?

Have students pair up, and give out drawing paper. Each pair of students is to design a recursive drawing. Display these so everyone can view the pictures. A nice follow-up to this activity would be to show the film *Maurits Escher, Painter of Fantasies* (Coronet Films).

Activity 5: Recursive Scripts

Play the tape recording called "Teacher's Example of Recursion." (Directions for creating this tape are given in the Materials section above.) After students have listened to it, show Transparency III with Transparency IV placed on top. Play the tape again, and this time point to the lines of the script. Tell students that this is a recursive tape recording because the recording contains copies of itself.

Remove Transparency IV so that only III is projecting. Explain to students that what they are now viewing isn't recursive. But if we were to place inside this script another copy of this script, then the result would be recursive. Take Transparency IV, lay it over III and demonstrate. Remove IV again.

Tell students that Transparency III is a pattern or template with a hole in it. We become involved in a recursive process if we attempt to fill in the hole with another copy of the original pattern (template). But there is still a hole to fill. If we put another copy of the original into that hole, there is still another hole to fill. This process could go on forever!

Now ask the three students who made the tape recording to explain to their classmates how the recording was produced. (See above.) These students will serve as helpers for other recording teams wishing to create recursive recordings. Point out that they can create any pattern script they wish as long as they leave a "hole" in it. This hole is to be filled by another copy of the original pattern—which itself has a hole in it that has been filled.

Helping Students with Recursion: Teaching Strategies

Part II: Moving to the Computer

by
Tim Riordon

[Editor's Note: This is the second of a three-part article. "Part I: Introducing Recursion with Non-Computer Recursive Experiences" was in the Dec./Jan. issue of TCT, Vol. 11 #5.
Permission to reprint is granted provided full credit is given the author and The Computing Teacher.*]*

Teacher Background

You can take three steps to prepare yourself for introducing Logo recursion to students.

- Set aside some time to work through Activities 7-13. Be sure to take time to do the suggested follow-up activities. Pay special attention to the technique for "mapping out" recursions.

"Your students can spend hours looking for number patterns and relationships which result from comparing the input numbers to the procedures and the resulting geometrical figures. When you involve your students in these explorations, a great deal of whole-class data gathering, organizing, hypothesizing and predicting is possible. Mathematics becomes a community enterprise."

- Familiarize yourself with the material presented in Chapters 7 and 8 of Watt's *Learning With Logo*. The material on variables is a prerequisite for being able to use recursion in all but the simplest ways. The POLY and INSPI explorations in Chapter 8 are important for three reasons: they provide practice with variables; they provide practice with recursion; and they form a rich mathematical microworld. Your students can spend hours looking for number patterns and relationships which result from comparing the input numbers to the procedures and the resulting geometrical figures. When you involve your students in these explorations, a great deal of whole-class data gathering, organizing, hypothesizing and predicting is possible. Mathematics becomes a community enterprise.
- Work through pages 32-37 of Abelson's *Logo for the Apple II*. Use the "mapping out" technique described in this article and apply it to the two examples given by Abelson. Map out COUNTDOWN 5 and TOWER 50 for practice. If you want to get a head start on your students, see if you can use the mapping-out technique to understand Abelson's MYSTERY procedure (pages 37-40). Try mapping out MYSTERY 3. (Part III of this article will contain the correct mapping.)

Materials

- Transparencies V-X; transparency marking pens.
- Recursive Play Scripts (one per student); butcher paper to draw design on; felt tip pen.

Activities

Activity 6: The POW Game

Use the top half of Transparency V to explain the recursive "POW" game to students.

```
TO POW
Whenever you hear the word POW do 3 things:
  RAISE ARM
  LOWER ARM
  SAY POW
END
```

Begin the game by saying POW. Point to each action your students are to do. When you point to the line SAY POW, remind them of the rule: Whenever you hear the word POW do 3 things. Play the game until they get the idea that it will go on forever. Tell them that the game is recursive: the description of the game contains its own name.

Tell students that a good way to think about what happens is as follows. Use a transparency marking pen and make the dotted arrow line pointing to the right:

```
TO POW
Whenever you hear the word POW do 3 things:
  RAISE ARM
  LOWER ARM
  SAY POW------>
```

Tell students that at the end of the line you are placing another copy of the procedure.

```
TO POW
Whenever you hear the word POW do 3 things:
  RAISE ARM
  LOWER ARM
  SAY POW------>  POW
                    RAISE ARM
                    LOWER ARM
                    SAY POW------>
```

It is important to use the word "copy" and to actually write in the copy. What you should *not* say is that when the procedure gets to the third line (SAY POW), it "goes back up" to the beginning. Avoid making any kind of a circular or looping motion with your hands. Now, start at the top line of the procedure and point to each line, demonstrating what happens:

This line tells us to raise our arms.

This line tells us to lower our arms.

This line *generates another copy of the procedure.* Trace your finger along the arrow until you come to the second copy of POW. Read each line of this copy:

This line tells us to raise our arms.

This line tells us to lower our arms.

This line *generates another copy of the procedure.* Quickly write in another copy and continue in this manner until students see that the process will go on forever.

Activity 7: Doing Dishes and Ongoing Processes

Have students pair up. Their task is to create a recursive procedure: TO DO DISHES.

```
TO DO DISHES
  WASH DISH
  ??????????
  ????????
END
```

Remind students that a recursive procedure must contain its own name. (Thus, DO DISHES must be a line somewhere in the procedure.) Let students share and revise their attempts.

Ask them to write a recursive TO EAT.

Activity 8: To the Computer with BOX

Now show students the old Logo standby: TO BOX.

```
TO BOX
  FD 50
  RT 90
  BOX
END
```

Let this procedure run until students observe that it will go forever. Then use the overhead or the blackboard to show students what is occurring. Begin by writing the procedure in the upper left corner of the screen. When you get to the recursive call (the line BOX), draw a dotted arrow to the right.

```
TO BOX
  FD 50
  RT 90
  BOX------>
END
```

At the end of the arrow, place a new copy of the BOX pattern. Your result will appear as follows:

```
TO BOX
  FD 50
  RT 90
  BOX------>  BOX
END             FD 50
                RT 90
                BOX
              END
```

Now continue this recursive process which we will call **mapping out.** It is important that your students see you do this, because a later activity will ask them to use this process to understand and predict the results of a recursive procedure. This process of "mapping out" will be an important heuristic device as students move into working with the more complex "embedded" recursive procedures. As you map out the process, use the phrase "generates another copy of the original pattern" each time you come to the recursive line, while making the dotted line and writing in the next copy.

Demonstrate what happens if you edit the procedure and change the input to the RT command. For example,

```
TO BOX
  FD 50
  RT 144
  BOX
END
```

Because this procedure no longer draws a figure resembling a box, many Logo users give a generalized version of this procedure the name

```
TO POLY :SIDE :ANGLE
  FD :SIDE
  RT :ANGLE
  POLY :SIDE :ANGLE
END
```

Now it's time for your students to begin experimenting with recursive graphics programs. You might begin by following Dan Watt's suggestions for exploring recursion with examples that do not make use of variables (see pages 88-94 of *Learning With Logo).* The clever cartoon on page 89 of Watt's book (showing a recursive procedure calling upon another copy of itself) should be reviewed with students. If your students have worked with variables, then additional exploration with various in-

puts to the POLY program could also be encouraged now. Invite your students to share their recursive creations. You can meet with each student individually to have him/her select and "map out" one recursive creation. This is an important check to help you evaluate whether your students have learned to use this important tool to understand how recursive procedures produce the results that they do. Transparency VI gives another example (PENTA) of a recursive procedure (a pattern) that has been mapped out.

The next activities presuppose that students have worked with variables. The power of recursion is hidden until people can combine these two powerful ideas. If your students have not yet worked with variables, it's time to take several weeks to thoroughly explore this area.

Activity 9: Recursive Procedures that Grow and Change

Show the procedure pattern at the top of Transparency VII.

```
TO POLY :SIDE
  FD :SIDE
  RT 144
  POLY :SIDE + 1
END
```

Demonstrate the design that results from teaching the computer this procedure and then running POLY 10. Try different input numbers so that students see how the input affects the design.

Next, demonstrate how this recursive procedure should be mapped out. Transparency VIII shows a 7-level mapping, but do not show this to students yet. It is valuable for students to observe the teacher simulating the computer—to see you start with the first instance of the procedure pattern and generate each successive copy of the original pattern with its particular input. Use the bottom part of Transparency VII to begin mapping out the procedure.

```
POLY 10
  FD 10
  RT 144
  POLY 10 + 1------>
END
```

Notice that this mapping out uses a specific input number that has been substituted in the place of the :SIDE of the procedure definition. This approach to mapping out recursive procedures avoids using the abstract Logo variable notation(:SIDE). Students see that each copy is based upon the original template. They also see that the input for each copy is derived from the previous copy. I believe this approach is easier to understand than the more general approach used by Abelson on pages 38, 39, 80, 89, 132, 135 and 137 of *Logo for the Apple II*. When you copy down the recursive line POLY 10+1 and draw the dotted arrow line to the right, tell your students: "This line generates another copy of the original pattern. (Point to the procedure pattern at the top of Transparency VII.) The input to this next copy is 11." Then add another copy to the mapping as follows:

```
POLY 10
  FD 10
  RT 144
  POLY 10 + 1------>  POLY 11
END                     FD 11
                        RT 144
                        POLY 11 + 1------>
                      END
```

Continue in this manner until you have mapped out six levels. This is enough to show how the star "grows" in the particular way that it does. If you wish, you may now show Transparency VIII containing the 7-level mapping. Use one finger to point to each line in the mapping while you draw the figure with a transparency marker. Demonstrate the one-to-one correspondence between each line in the graphic and each line in the mapping out of the recursion.

Follow up this demonstration by having students experiment with this POLY procedure. Encourage them to change the angle number. For example,

```
TO POLY1 :SIDE
  FD :SIDE
  RT :160     (or RT 90 or 60 or 108)
  POLY1 :SIDE + 1
END
```

If after several days no one has happened upon the following, suggest that they try changing the angle number so that it is very close to—but not equal to—the numbers suggested above. For example,

```
TO POLY 2 :SIDE
  FD :SIDE
  RT 143        (or 142 or 145 or 146)
  POLY2 :SIDE + 1
END
```

Spiral patterns are formed. This suggests another project—how to make just the spiral arms appear (like the pictures of galaxies)! Of course, encourage your students to share their results and discoveries.

Activity 10: Letting Angles Grow or Decrease

Demonstrate this procedure and then map out the result of running INSPI 0.

```
TO INSPI :ANGLE
  FD 10
  LT :ANGLE
  INSPI :ANGLE + 30
END
```

Map the result for 7 or 8 levels. (You'll need to write small or use a large piece of butcher paper!) The reason for mapping this out is twofold. First, it is important for students to see and hear many ex-

amples of how a recursive procedure generates copies of itself. It is especially important for them to see how to handle these mappings when variables are involved. The second reason for mapping this one out is because its behavior is so strange. For example, compare the results of running INSPI 0 vs. INSPI 5. If you have a printer, encourage some students to systematically explore the patterns which result from various inputs to INSPI procedures. In the above example, the :ANGLE was increased by 30 each time. This number—called an increment number—was selected to make it easier for you to map out the recursion for your class. Encourage students to try INSPI procedures with different increment numbers.

Activity 11: Taming Recursion with Conditionals

Soon after your students begin working with recursion, you'll be confronted with the question, "Is there any way to stop it?" This is a good time to introduce students to the use of conditionals to "tame recursion."

Show students the design at the top of Transparency IX and teach the computer the procedure.

```
TO NESTED.BOXES :SIZE
  SQUARE :SIZE
  MOVE
  NESTED.BOXES :SIZE + 20
END
```

This procedure will not work until you teach the turtle TO MOVE and TO SQUARE :SIZE. After creating these subprocedures, demonstrate the result of running NESTED.BOXES 5. Try running the procedure with some other inputs. Explain that you are now going to show how to stop the procedure from growing indefinitely. Using a transparency marking pen, place the following line in the procedure pattern at the top of Transparency IX.

```
IF :SIZE > 80 STOP
```

In Apple Logo—

```
IF :SIZE > 80 [STOP]
```

The result will now look like this:

```
TO NESTED.BOXES :SIZE
  IF :SIZE > 80 STOP
  SQUARE :SIZE
  MOVE
  NESTED.BOXES :SIZE + 20
END
```

Run the revised version and show what happens. Ask your class, "Can you suggest a change that will allow the procedure to stop when the box has a side of 100?"

Next, map out the result of running NESTED.BOXES 5 so students can see how each successive copy of the procedure has a conditional line **but with a different 'IF...' statement!** Here is how the first few levels of the mapping will appear:

```
NESTED.BOXES 5
  IF 5 > 80 STOP
  SQUARE 5
  MOVE
  NESTED.BOXES 5 + 20 ->
END
```

```
NESTED.BOXES 25
  IF 25 > 80 STOP
  SQUARE 25
  MOVE
  NESTED.BOXES 25 + 20 ->
END
```

```
NESTED.BOXES 45
  IF 45 > 80 STOP
  SQUARE 45
  MOVE
  NESTED.BOXES ... > etc.
END
```

It's important to point out how each successive copy of the procedure gets its input value from the previous copy. Once this input is determined, you simply look at the procedure pattern at the top of Transparency IX and substitute this input at each and every spot where :SIZE appears in the original pattern. This approach of mapping out recursions will pay dividends when we encounter embedded recursion in Part III.

Follow up this demonstration by having students use conditionals to tame some of the recursive procedures they have worked with before. Point students toward some of the explorations suggested in Chapters 7 and 8 of Watt's *Learning With Logo*. Students will enjoy his cartoons.

Activity 12: Recursive Computer Drama

Molly and Dan Watt suggest having students act out "recursive scripts." Here is one possibility for such an activity.

Give out 15 Recursive Play Scripts (see Figure 1) and ask the student actors to familiarize themselves with their parts. The other students will be the audience. Later they may have their turn. Each actor also gets a blank index card upon which to write an input number. Place a large sheet of butcher paper on the floor with the felt tip pen at the center. Explain that the pen is the turtle which students are to control when it is time for their part to be played.

Ask your actors whether they have any questions about their job. If not, have them line up "OFF STAGE." The play is about to begin. Announce to the audience that they are about to see a performance of a new play called HEX 26. Move off stage and tell the first actor to begin. Give the actor an index card with the input number of 26.

When the play is finished, ask students why only ten actors got to control the turtle. Can they think of two changes that would permit more actors to be involved? (An initial input lower than 26 or changing the conditional line in the script.)

Ask the audience whether they would like to perform the same play or make up one of their own. Before your class moves on to a different recursive play, show Transparency X—a mapping of HEX 26. Ask the actors to pretend that this is the script and to find the "lines" which they acted.

Finally, it's worth commenting upon Step 8 in the Recursive Play Script. This step reinforces the idea that when a recursive procedure STOPs or ENDs, control reverts back to the calling procedure. Even though the actors have nothing further to do with the turtle, **they are not really finished until they have reported back to their respective calling procedures.**

Figure 1.
RECURSIVE COMPUTER DRAMA
(Script)

When you are called to perform your job, do the following things:

1) Take the other person's place.
2) Remember who called you.
3) Get an input card from the person who calls you.
4) Check to see if your input number is bigger than 35. IF IT IS BIGGER THAN 35, DON'T DO ANY MORE STEPS. TELL THE PERSON WHO CALLED YOU THAT YOU ARE FINISHED.

Do the rest of the steps only if your input was smaller than 36.

5) Go forward your input number and draw your trail behind you. Use a ruler to make your line straight.
6) Turn right 60 degrees.
7) Think of your input number and add 1 to it. Write this new number on a blank index card and give it to the next person waiting to be called.
8) Wait off stage until someone tells you that he or she is finished. When you get that message, pass it on to the actor who first called you. If no one called you, walk out and say, "THE END."

TRANSPARENCY V

TRANSPARENCY VI

BEST COPY AVAILABLE

BEST COPY AVAILABLE


```
VII      TO POLY :SIDE
          FD :SIDE
          RT 144
          POLY :SIDE + 1
         END
```

TRANSPARENCY VII

```
VIII     TO POLY :SIDE   (pattern)
          FD :SIDE
          RT 144
          POLY :SIDE + 1
         END
```

MAPPING OF POLY 10

TRANSPARENCY VIII

```
IX       TO NESTED.BOXES :SIZE
          SQUARE :SIZE
          MOVE
          NESTED.BOXES :SIZE + 20
         END
```

MAPPING OF NESTED.BOXES 5

TRANSPARENCY IX

```
X        TO HEX :SIDE          (pattern)
          IF :SIDE > 35 STOP
          FD :SIDE
          RT 60
          HEX :SIDE + 1
         END
```

MAPPING OF HEX 24

TRANSPARENCY X

END

Helping Students with Recursion: Teaching Strategies

Part III: Teaching Students About Embedded Recursion

by
Tim Riordon

[Editor's note: This is the third of a three-part article. Parts I and II were in the December/January and February issues of TCT. *Permission to reprint is granted, provided full credit is given the author and* The Computing Teacher.*]*

TEACHER BACKGROUND

Understanding and using embedded recursion comes later in one's Logo career. Some of the following lessons may be appropriate for a large group of students, but it is highly unlikely that entire classes would be ready at the same time for activities like 19 and 20, which involve recursive examples with numbers and with words. Nevertheless, these activities are included because they will help the teacher who has two or three students who are reaching for this powerful but sophisticated tool. The best preparation for teaching these activities is to work through them yourself. Then try working through the examples provided in the Abelson manual.

MATERIALS

- Transparencies I-IV; Transparencies XI-XVI

ACTIVITIES

Activity 13: Providing Students with Intuitions About Embedded Recursions

Have your students re-do Activities 3, 4 and 5 from Part I. Emphasize the idea of objects containing copies of themselves.

Activity 14: Predicting Embedded Recursions

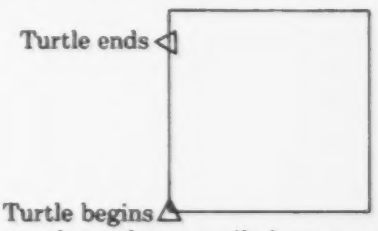
Explain to your students that you are going to show them some procedures that are recursive, but in these procedures the recursive line does not occur at the very end of the procedure. Ask students to predict what the computer screen will show as a result of running MYSTERY 4 and DESIGN 1 which follow:

```
TO MYSTERY :NUMBER
  IF :NUMBER = 0 STOP
  PRINT :NUMBER
  MYSTERY :NUMBER - 1
  PRINT :NUMBER
END
```

```
TO DESIGN :COUNT
  IF :COUNT > 4 STOP
  FD 50
  RT 90
  DESIGN :COUNT + 1
  FD 40
  LT 90
END
```

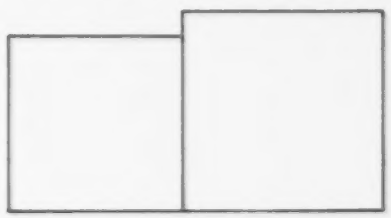
Ask whether anyone predicted these results.

```
4
3
2
1
4
3
2
1
```



Run the procedures to show what actually happens.

```
4
3
2
1
1
2
3
4
```



Review with students how to map out recursive procedures in order to accurately predict what will occur. Use the mapping for MYSTERY 4 on Transparency XI as an example. Walk your students through this example. Then ask them to map out DESIGN 1 on their own. Check their maps. Show the message at the bottom of Transparency XI and show how this rule applies in each of these cases. (The rule is that when a recursive procedure generates a copy of itself, the *called procedure* must finish all of its business, and then control returns to the *calling procedure*, which can then proceed with its unfinished business.) Ask them to trace the flow of control through the various levels, drawing and turning as the turtle will turn. As you walk students through the maps of these two recursive procedures, identify in each case when a procedure is "tabling some business until later." Make certain that your students are very clear about what is happening before going on to the next activity. Here are two other short procedures; have students map them out and make predictions before they run them to check their results.

```
TO CIR
FD 5
  RT 30
  IF HEADING = 0 STOP
  CIR
  FD 5
  LT 30
END

TO TRI :TERM
  IF :TERM = " STOP
  MAKE "MESSAGE SE :MESSAGE FIRST
    :TERM
  PRINT :MESSAGE
  TRI BUTFIRST :TERM
  PRINT :MESSAGE
END
```

Ask students to predict what will be printed out if they use their own names as the input to the TRI procedure. But note: Before trying to run one of these procedures, the following line must first be typed in:

```
MAKE "MESSAGE [ ]
```

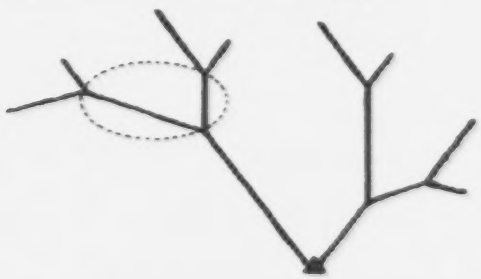
The second line in the TRI procedure cannot work unless the name "MESSAGE has already been created. Also, the input to this procedure is supposed to be a word; it should be quoted. Examples: TRI "BARBARA or TRI "JOSHUA.

Activity 15: Seeing Patterns and Processes

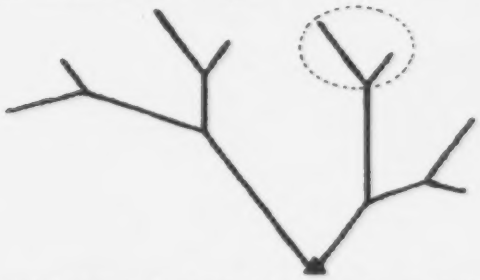
Have students work in pairs for this activity. One at a time, uncover the patterns on Transparency XII. Ask students to talk about whether they can see a pattern or systematic process. In each case, give students plenty of time to talk about what they see.

Activity 16: Graphic Designs That Contain Embedded Copies of Themselves

Show students the binary tree labelled TREE1 on Transparency XIII and remind them of the fundamental pattern. (The basic pattern is a "vee"-shaped branch having a left side that is twice as long as the right side. The angle between the sides is 70 degrees.) With a transparency marking pen, circle one of the "basic units." For example:



It is important that your students see that this basic unit is embedded throughout the entire design. Here is the basic unit circled at a different place in the entire design.



Explain to students that in order to create the entire binary tree design—in fact, in order to create many recursive graphic designs—they need to know only two things:

1) The turtle's place and heading just prior to any recursive call; and
2) How to draw the basic pattern, using variables, beginning from and ending where the turtle is just prior to the recursive call.

Referring to the branch you have just circled on the transparency, ask students to say where the turtle is just prior to this branch. (The turtle is at the base of the "vee" and is pointing in a direction which goes between the branches of the "vee.") Show students that this would be true no matter which "vee"-branch unit you circled.

Now ask students to help you create a "vee"-branch unit starting from the turtle's location and heading just prior to drawing the branch. Here is the procedure you want to elicit. Because you will be referring to this later, it's best to write this on the board:

```
TO TREE1 :SIZE
  LT 35
  FD :SIZE
  BK :SIZE
  RT 70
  FD :SIZE/2
  BK :SIZE/2
  LT 35
END
```

Before going on, check to see that students understand where the various angle inputs come from. Then run the procedure to see that it works.

Explain that once someone has created the fundamental unit, only two things are needed to turn the procedure into one which will make the entire binary tree:

- Copies of the pattern just created must be embedded at the correct places; and
- A conditional STOP rule must be placed in the procedure.

Refer again to TREE1 on Transparency XIII and point out that each "vee"-branch unit is 2/3 as big as the branch unit it grows out of. Now place your

transparency marking pen at the base of the tree (near the point of the biggest "vee"-branch unit). Ask a student to read each line of the procedure you have written so far while you trace out what the turtle will do. Tell students that your purpose is to discover exactly where you need to embed recursive calls to TREE1 :SIZE*2/3.

Make sure that students see that the proper place to embed the recursive calls are just after the turtle has made its FORWARD movements and just before it makes its BACKWARD movement. When the turtle moves forward, it puts on hold the completion of the "vee"-branch unit it is working on and starts making another "vee"-branch unit, 2/3 as big. This is true for both the right side of the branch and the left side of the branch. So, two recursive calls must be embedded. Modify the procedure you have written on the board.

```
TO TREE1 :SIZE
  LT 35
  FD :SIZE
TREE1 :SIZE*2/3
  BK :SIZE
  RT 70
  FD :SIZE/2
TREE1 :SIZE*2/3
  BK :SIZE/2
  LT 35
END
```

Make sure that students see that what you have done is to "embed a copy of the procedure within itself." Use these exact words to describe what you have done. Type this new version of TREE1 into the computer and run TREE1 96. (Because there is no STOP rule, the procedure will not generate anything more than the left-most set of branches for the binary tree. Take the time to map out what is happening so that students fully understand why the entire tree is not generated. The mapping will appear as follows. Make sure that students see that the lack of a STOP rule means that each successive copy of the original pattern starts making a branch, but has to suspend its work and never gets a chance to complete the job.

```
TREE1 96
  LT 35
  FD 96
TREE1 96*2/3 -> TREE1 64
  BK 96           LT 35
  RT 70           FD 64
  FD 96/2       TREE1 64*2/3 -> TREE1 42.6
TREE1 96*2/3      BK 64             LT 35
  BK 96/2         RT 70             FD 42.6
  LT 35           FD 64/2         TREE1 42.6*2/3 ->
END             TREE1 64*2/3        BK 42.6
                  BK 64/2           RT 70
                  LT 35             FD 42.6*2/3
                END               TREE1 42.6*2/3
                                    BK 42.6/2
                                    LT 35
                                  END
```

Insert the conditional STOP rule, IF :SIZE < 29 STOP as follows.

```
TO TREE1 :SIZE
  IF :SIZE < 29 STOP
  LT 35
  FD :SIZE
TREE1 :SIZE*2/3
  etc.
  etc.
```

Run the procedure with the input 96. Use Transparency XIII to walk students through a mapping of TREE1 96.

Follow this demonstration up by giving students several class sessions to explore binary trees. Encourage them to change some of the angle turns or some of the factors by which the :SIZE input is multiplied. They should print the resulting trees and include with each picture the exact procedure that produced the tree.

Activity 17: More Graphics Examples of Embedded Recursion

Have students share their explorations with the binary trees. Next, pair students up and show Transparency XIV. Ask students to identify the "basic unit" and see if they can write the procedure that will produce the entire recursive design. List the following suggestions as hints.

1) Draw the basic unit.
2) Note the turtle's heading and location just before the basic unit is executed.
3) Write a procedure for the basic unit using variables with the turtle starting and finishing in the same place (and heading as it began).
4) Embed copies of the procedure at the proper places. Be sure to multiply the :SIZE variable.
5) Put in a conditional STOP rule.

Encourage students to share "bugs" with each other. Some bugs lead to unexpected but interesting results. Encourage students to print out unusual results and to include copies of the program which generated the particular design. These will make good discussion pieces—chances for others to try to figure out just how the program created the design that resulted. Have students map out at least one of the recursive design procedures and talk you through it. This will be your way of knowing whether they have mastered the mapping out technique.

Activity 18: A Numerical Example

Present the following task: to write a procedure that will raise any number to a given power. The procedure should require two inputs: the number and the exponent. This will be a procedure that processes these two inputs and outputs a single number—the answer. We'll call the procedure TO POWER :NUMBER :EXPONENT. Thus, POWER 5 2

outputs 25 (the 2nd power of 5). POWER 5 4 outputs 625 (because 5*5*5*5 equals 625).

Lead students through the following series, making sure that they see the truth of each step. Tell them that you are going to transform the problem of raising 5 to the 4th power into a series of simpler problems.

Ordinary Language	Computer Language
Goal: What is 5 to the 4th power?	POWER 5 4
Step 1) 5 to the 4th power is the same as 5* (5 to the 3rd power)	5* POWER 5 3
Step 2) 5* (5 to the 3rd power) is 5*5* (5 to the 2nd power)	5*5* POWER 5 2
Step 3) 5*5* (5 to the 2nd power) is 5*5*5* (5 to the 1st power)	5*5*5* POWER 5 1
Step 4) 5 to the 1st power is 5	POWER 5 1 is 5
Thus, 5 to the 4th power is 5*5*5*5 or 625	POWER 5 4 is 5*5*5*5=625

Point out to students the pattern of descending numbers:

POWER 5 4 POWER 5 3 POWER 5 2 POWER 5 1

which stops when POWER 5 1 is reached.

Now use Transparency XV, which gives the procedure for producing the power of a number. The bottom part of the transparency shows a mapping for POWER 6 3. Walk students through this mapping. Then ask students to work together to make a mapping of POWER with other inputs. Be sure to point out that the STOP rule uses OUTPUT instead of STOP. Use the mapping to explain that when the process of transforming the problem finally stops, there must be a number present to multiply.

If some students want to try to apply this approach to another mathematical example, suggest that they create a procedure TO FACTORIAL :NUMBER which behaves as follows:

Example	Result		
FACTORIAL 4	24	i.e.	(4*3*2*1)
FACTORIAL 7	5040	i.e.	(7*6*5*4*3*2*1)

Tell students that the procedure is very similar to the POWER procedure. If they get stuck on the STOP rule, it is IF :NUMBER = 1 OUTPUT 1.

Activity 19: Examples With Words

The approach just illustrated with numbers has many applications in working with words and lists of information. Frequently, when the task is to process a list or string of information according to some rule, the process can be accomplished by creating a new list or word which is the result of combining two objects. The first object is the result of applying the rule to the first item in the input word or list. The second object is what is output by a recursive application of the rule to all but the first item of the original input word or list. Sound confusing? Re-read it and see how these statements apply to the POWER 5 4 example above. After completing the following activity, re-read the statement and see how it applies to the example below.

Suppose the task were to take a name and put it into some secret code. We want to create a procedure TO ENCODE :TERM which will take single words as inputs and produce an encoded version of that term. We'll assume that we already have a procedure which will take a single letter as input and ouput the code letter. This procedure might be as follows:

```
TO GCL :LETTER
IF :LETTER = "A OUTPUT "R
IF :LETTER = "B OUTPUT "Z
IF :LETTER = "C OUTPUT "P
PRINT [ HAVEN'T GOT ALL THE LETTERS ]
OUTPUT "
END
```

(This stands for Get Code Letter!)
(R is the code letter for A)
(Z is the code letter for B)
(P is the code letter for C)

We would like our procedure ENCODE to take a term like "CAB and output the encoded version "PRZ. Show Transparency XII. Some light bulbs will probably light up. In order to ENCODE a whole term (for example "CAB), we need to create a new WORD which will consist of two things: the code letter for the first letter in our term (in this case the code letter for "C); and the result of ENCODING the remainder of the term (that is, whatever results when you ENCODE "AB).

Refer to Transparency XVI which contains the procedure and a mapping for ENCODE "ABCC. The Logo primitive WORD takes two inputs—the two objects mentioned above. These have been put inside parentheses for clarity. Walk students carefully through this mapping. A follow-up activity for those students who understand would be to create a mapping for the REVERSE procedure given by Abelson on page 130 of *Logo for the Apple II*. This procedure is difficult for many people. The mapping technique suggested in this article will help students understand how the procedure reverses a term. Students might use their own names as inputs.

FINAL REMARK

I hope that readers will find these activities helpful. Several people have asked me a question which I think about from time to time. Why bother to teach recursion? It's a difficult concept. Are there really any problems that, when subjected to a programmer's scrutiny, are "naturally" solved by recursion? I invite readers to share two sorts of ideas: What are some problems that lend themselves to recursive solutions? What are other examples of recursive experiences that we can point out to our students? We'll share your ideas with the rest of the Logo community. END■

Computers and the Humanities in Schools and Colleges

West Chester, PA June 28-29, 1984

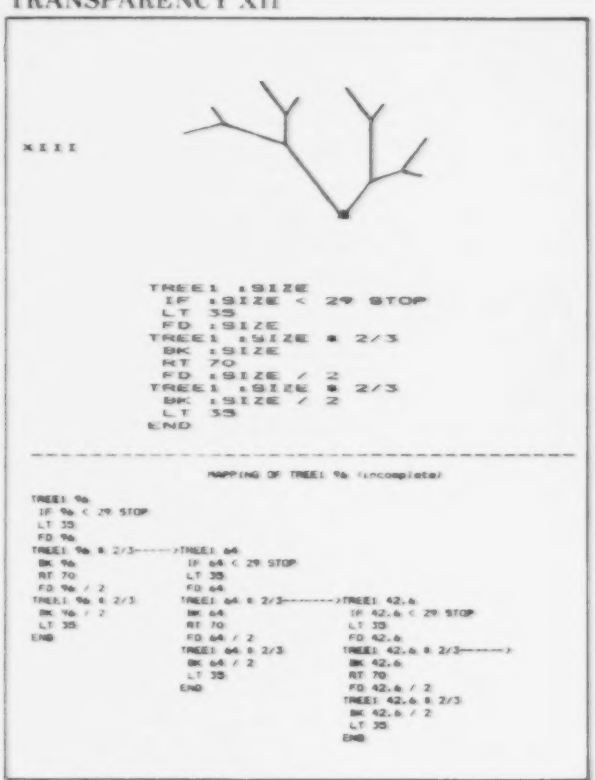
Sessions will focus on computer applications at all educational levels and on ethical implications for schools and society. Some sessions will require familiarity with computer terminology and use. A chief goal of the conference is to enhance humanities instruction in local schools and colleges.

For more information, contact Kostas Myrsiades or Robert Weiss, Department of English, West Chester University, West Chester, PA 19383, (215) 436-2745 or 2281.

XII
WHAT PATTERNS DO YOU SEE?
(B I R D)
D (B I R)
D R (B I)
D R I (B)
D R I B ( )
ENCODE [HERE IS THE MESSAGE]
3727 ENCODE [IS THE MESSAGE]
3727 86 ENCODE [THE MESSAGE]
3727 86 137 ENCODE [MESSAGE]
3727 86 137 4766987 ENCODE [ ]
HOW MUCH IS 5#, PROVIDED THAT:
5# = 5 * 4#
4# = 4 * 3#
3# = 3 * 2#
2# = 2 * 1#
1# = 1
Could you find 7# ?

TRANSPARENCY XII

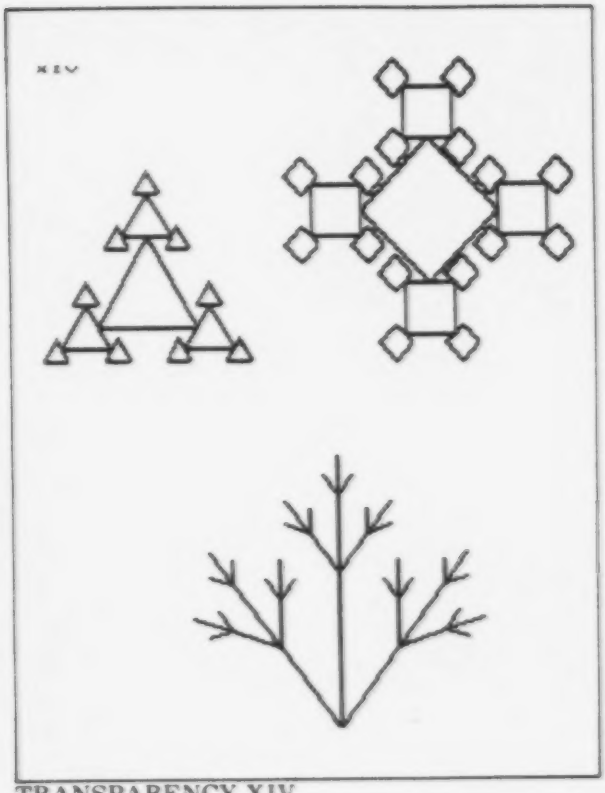
TRANSPARENCY XI



TRANSPARENCY XIII

BEST COPY AVAILABLE

BEST COPY AVAILABLE



TRANSPARENCY XIV

XV
TO POWER :NUM :EXP
IF :EXP = 1 OUTPUT :NUM
OUTPUT :NUM * POWER :NUM :EXP - 1
MAPPING OF POWER 6 3
POWER 6 3
IF 3 = 1 OUTPUT 6
OUTPUT 6 * POWER 6 3-1
POWER 6 2
IF 2 = 1 OUTPUT 6
OUTPUT 6 * POWER 6 2-1
POWER 6 1
IF 1 = 1 OUTPUT 6
OUTPUT 6 * POWER 6 1
END

TRANSPARENCY XV

XVI
TO ENCODE :STUFF
IF :STUFF = "" OUTPUT ""
OUTPUT WORD (BCL FIRST :STUFF) (ENCODE BUTFIRST :STUFF)
END
(BCL is a procedure that takes a letter and looks up the code letter.)
MAPPING OF ENCODE "ABCC

TRANSPARENCY XVI END

[Editor's Note: Readers who wish full-sized, black on white masters for the entire set of 16 transparencies may order them by taking the following steps:

1. *Put 37¢ postage and your mailing address on a 9x12 clasp envelope.*
2. *Send the envelope and $2.00 to Tim Riordon, Alpine Educational Computing Services, 1633 High St., Eugene, OR 97401.]*

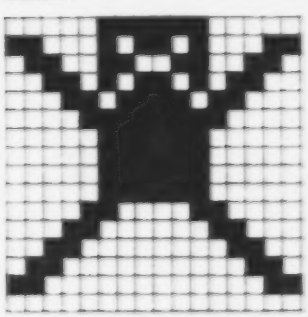
THE LOGO CENTER

Kathleen Martin • Tim Riordon

So What About Sprites?

Yes, what about sprites? You can be sure that we have not forgotten them. Tim and I purposely chose to reserve the last two issues of the column (combined in this issue!) for work with sprites. The choice is based on the belief that children should accumulate some experience with turtle graphics before engaging sprites. Such experience enables the child to direct the progress of the sprites rather than becoming mesmerized by their arcade-like movement around the screen. The sprite is simply another kind of "object to think with." While some of the characteristics of the sprite and the turtle are different, the kind of thinking that governs programming both is much the same. The important differences are that the shape of the sprite can be changed, and it can be set in motion.

To illustrate some of the projects that emerge when children work with sprites, we'll begin by describing a game that was developed by 8-year-old Scott. The work was done using TI Logo. (Other Logo versions use similar commands.) The game did not start as a game, but as a kind of fooling around. Like most children, Scott thoroughly enjoys designing his own shapes. Shape-making is a relatively easy technique that children quickly master using the MAKESHAPE command (abbreviated MS). Since the first five shapes are reserved in TI Logo, Scott used the shape number 6. When the MAKESHAPE command is entered (MS 6), the shape editor presents a 16x16 grid. Using the arrow keys and the function key, Scott designed the shape below.



MAKESHAPE 6

Figure 1.

A sprite is invisible until assigned a shape, color and position on the screen. Scott entered the following set of commands to make the sprite appear.

```
TELL 1
CARRY 6
SETCOLOR :RED
HOME
```

He then began moving the sprite, now a creature named "Ralph," around the screen using the SETSPEED and SETHEADING commands. Children who have worked extensively with turtle turns have little difficulty adjusting to headings, especially if both are taught with reference to the clock. Scott soon decided that he wanted to control the sprite creature using the keys instead of typing in commands, so he wrote the following two procedures.

```
TO S
SETHEADING 270
CLEARSCREEN
END
```

```
TO D
SETHEADING 90
CLEARSCREEN
END
```

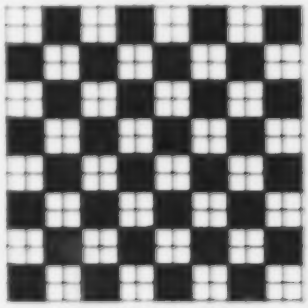
The experience of writing procedures in the turtle mode transfers easily to sprites. Now that Scott could move the sprite from side to side using the S and D keys (and ENTER), he decided to use additional keys to change the sprite's speed.

```
TO R
SETSPEED 20
CLEARSCREEN
END
```

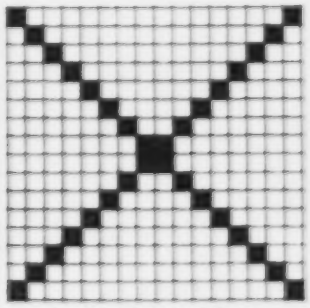
```
TO T
SETSPEED 50
CLEARSCREEN
END
```

```
TO Y
SETSPEED 100
CLEARSCREEN
END
```

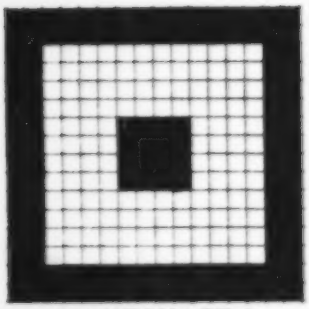
Now that the sprite creature had the possibility of two directions and three speeds, it needed some objects to evade. Here Scott's fooling around shifted to a conscious game design. Scott made another five shapes, assigning them the numbers 7, 8, 9, 10 and 11.



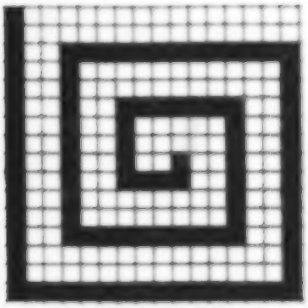
MAKESHAPE 7



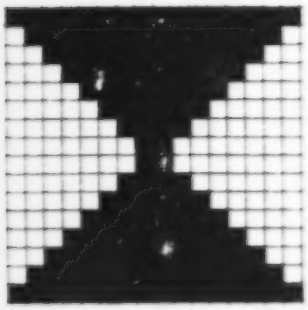
MAKESHAPE 8



MAKESHAPE 9



MAKESHAPE 10



MAKESHAPE 11

Figure 2.

Scott decided that he wanted these five shapes to move down from the top of the screen. He did not know how to assign positions using the x and y coordinates, a skill necessitating an understanding of negative numbers. Instead he devised his own way of moving sprites to the place that he wanted them. First he made the sprite appear on the screen by assigning shape, color and the home position to a sprite. Then he simply set the speed of the sprite at 1 and assigned it a heading of 90 or 270. The sprite would then slowly move toward the right or left side of the screen. When the sprite arrived at the desired position, Scott temporarily set its speed at 0, changed its heading to 180, and assigned the sprite the new speed that he wanted. In this way he managed to get the five shapes raining down from the top of the screen at different speeds. Below is an example of the sequence of commands he used.

```
TELL 2
CARRY 7
SETCOLOR :GREEN
HOME
SETHEADING 270
SETSPEED 1
SETSPEED 0          (When sprite arrives in
SETHEADING 180       desired position.)
SETSPEED 70
```

Now Scott's game was complete. He had only to avoid the avalanche of shapes by using the keys (really procedures) S, D, R, T and Y. There is one problem, however. The procedures are responded to by the last sprite addressed unless otherwise specified within the procedures themselves. Consequently, Scott had to go in and modify his five procedures to address sprite 1. (Abbreviations are used.)

```
TO S      TO D      TO R      TO T      TO Y
TELL 1    TELL 1    TELL 1    TELL 1    TELL 1
SH 270    SH 90     SS 20     SS 50     SS 100
CS        CS        CS        CS        CS
END       END       END       END       END
```

Children seem to particularly enjoy using sprites to construct simple games such as the one that Scott made. Let's look at a game that introduces the WAIT command. WAIT is a primitive in some Logo versions (e.g., TI and Apple) but needs to be defined in other versions.

```
TO WAIT :TIME
IF :TIME = 0 STOP
WAIT :TIME - 1
END
```

Begin this game by having the turtle draw a horizontal line across the screen. Use the MAKESHAPE command to design a creature that will stand on the line and attempt to jump over objects that move toward it from the side of the screen. There are two ways to locate the creature on the line. One way is to use the same approach chosen by Scott in his game, i.e., HOME, SETHEADING 180, SETSPEED 1, SETSPEED 0 (when the creature hits the line). The second way is to introduce the commands for setting coordinates—in this case, SY.

Design a shape (anything will do), and start it moving across the screen parallel to the line and toward the creature. This is again done using the commands for setting coordinates or Scott's alternative. Now it is necessary to write a procedure to make the creature jump when a key is pressed, thus avoiding the objects directed at it. Changing coordinates with a WAIT command between is one way. Changing headings with a WAIT command between is another.

```
TO E
TELL 1
SY 35
WAIT 20
SY 15
END
```

```
TO E
TELL 1
SETHEADING 0
SETSPEED 30
WAIT 25
SETSPEED 0
SETHEADING 180
SETSPEED 30
WAIT 25
SETSPEED 0
END
```

When the E key is pressed and entered, the creature will appear to jump. This game can be modified to allow the creature to jump from level to level with objects hurtling at him on each level. The creature can also jump while moving.

In introducing children to sprites it is sometimes helpful to assign a task for them to complete. For example, you can ask them to use the turtle to draw a path for a sprite and then to make the sprite follow it. The procedure PATH draws a path.

```
TO PATH
TELL TURTLE
HOME
REPEAT 4 [ FD 70 RT 90 ]
PU FD 15 RT 90 FD 15 LT 90 PD
REPEAT 4 [FD 40 RT 90 ]
HT
END
```

READY positions a sprite carrying the shape of a ball in the corner of the path.

```
TO READY
TELL 1
CARRY :BALL
SETCOLOR :RED
SX 0
SY 15
END
```

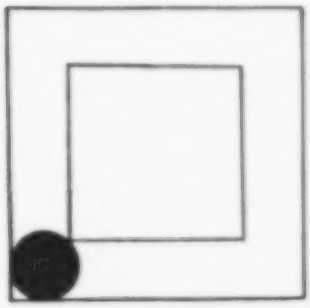


Figure 3.

PRACTICE is a procedure used to determine an appropriate combination of time and speed to move the ball from the bottom left corner to the top left corner. If SETSPEED is 20, then a WAIT of 45 works well.

```
TO PRACTICE
SETSPEED 20
WAIT 45
SETSPEED 0
END
```

It may be necessary for the children to try a variety of combinations before hitting on one that works. Once a combination has been found, the GO. AROUND procedure can be written.

```
TO GO.AROUND
SETHEADING 0
SETSPEED 20
WAIT 45
SETHEADING 90
WAIT 45
SETHEADING 180
WAIT 45
SETHEADING 270
WAIT 45
SETSPEED 0
END
```

More advanced children can use the MAKE command and a variable to simplify the above procedure.

```
TO GO.AROUND :H
SETSPEED 20
REPEAT 4 [ SETHEADING :H WAIT 45 MAKE
  "H :H + 90 ]
SETSPEED 0
END
```

All of the procedures (except PRACTICE) can then be combined into a single procedure.

```
TO RUN.TRACK
PATH
READY
WAIT 30
GO.AROUND
END
```

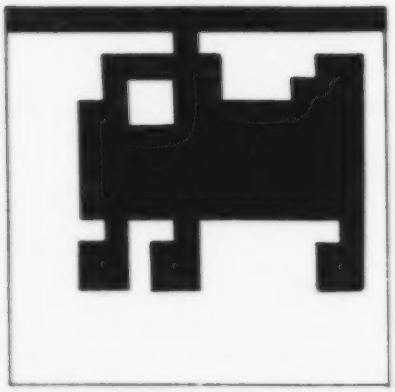
Another task that constitutes good practice for the children involves hitting a target with two (or three) sprites simultaneously. Use the turtle to draw the target. Locate the sprites at different positions on the screen. The challenge is to assign them speeds and headings that will make them arrive at the target at the same time.

The activities that have been described this month constitute an introduction to sprites for younger children. Advanced students have little difficulty incorporating sprites into their procedures. Now we will demonstrate some ways that teachers can use sprites in teaching traditional concepts to younger children. END■

More About Sprites (and the Turtle too)

In the last Logo Center, we suggested a few ways that you might introduce children to sprites. This time we'll present several models of interactive sprite programs which can be expanded, truncated or modified to serve your individual classroom needs. These aren't programs that children would write, but programs that teachers write for children. TI Logo is used in these activities.

Since you might use this first program to explore a little geography, we'll call it EXPLORING. But first we need something in which to travel. A helicopter is a versatile mode of transportation, so let's use MAKESHAPE to design one. Type MS 6 to get an empty grid on the screen and then shade it like the one below (or however you wish).



EXPLORING consists of two procedures, SETUP and FUEL.

```
TO EXPLORING
SETUP
FUEL 0
END
```

We'll use a SETUP that assigns a shape, color and position to get the helicopter on the screen. Since we'll be needing the Turtle later in the procedure, we'll also include it in the SETUP.

```
TO SETUP
TELL TURTLE HT
TELL SPRITE 1
CARRY 6
SETCOLOR :RED
HOME
END
```

FUEL is a recursive procedure that allows the helicopter to fly.

```
TO FUEL :SPEED
SETSPEED :SPEED
COMMAND
FUEL :SPEED
END
```

The helicopter moves in whatever direction and at whatever speed is requested through the COMMAND procedure.

```
TO COMMAND
MAKE "COM READWORD
IF :COM = "NORTH SETHEADING 0
IF :COM = "SOUTH SETHEADING 180
IF :COM = "EAST SETHEADING 90
IF :COM = "WEST SETHEADING 270
IF :COM = "FASTER MAKE "SPEED :SPEED + 5
IF :COM = "SLOWER CHECK
IF :COM = "STOP MAKE "SPEED 0
IF :COM = "GO MAKE "SPEED 5
END
```

The CHECK procedure tests the speed of the helicopter. If the speed is less than 5 and SLOWER is typed, the procedure stops the helicopter. The CHECK procedure is necessary to prevent the helicopter from moving backwards.

```
TO CHECK
TEST :SPEED < 5
IFTRUE SETSPEED 0
IFFALSE MAKE "SPEED :SPEED - 5
END
```

READWORD is a procedure that waits for a command to be typed, clears any text off the screen, and then allows the selected command from within COMMAND to be executed.

```
TO READWORD
PRINT [ ]
PRINT [ ]
PRINT [ ]
PRINT [ ]
PRINT [ ]
OUTPUT FIRST READLINE
END
```

Now that the helicopter is fueled and ready to take off, it needs someplace to fly. You can map the screen in any number of ways. A transparency of the United States with specific cities labeled is one possibility. Cutouts of the continents is another. The children can work out their own flight plans or be given a designated itinerary. EXPLORING can be integrated with research projects that encourage the children to learn more about the places that they land. Other directions can be added to the procedure if desired.

```
IF :COM = "NORTHEAST SETHEADING 45
IF :COM = "SOUTHEAST SETHEADING 135
IF :COM = "SOUTHWEST SETHEADING 225
IF :COM = "NORTHWEST SETHEADING 315
```

To reduce typing it might be well to use abbreviations for the various directions: N, S, E, W, NE, SE, SW, NW.

If you don't have sprites available, EXPLORING can be modified easily to travel by way of the Turtle.

```
TO FUEL :DISTANCE
PU
FD :DISTANCE
COMMAND
FUEL :DISTANCE
END
```

Simply go through the COMMAND procedure and change SPEED to DISTANCE. (You might also want to change the increment of 5 to an increment of 1.) Of course, none of the sprite information is necessary in SETUP.

The same program used to explore the geography of the screen can also be used to explore spatial relations. Let's add procedures to SETUP which allow us to draw some geometric shapes on the screen around which the helicopter (or Turtle) can meander.

```
TO TRIANGLE
PU SXY -30 0 SH 0 PD
RT 30
REPEAT 3[ FD 50 RT 120 ]
END
```

```
TO SQUARE
PU SXY -40 30 SH 0 PD
REPEAT 4[ FD 40 LT 90 ]
END

TO CIRCLE
PU SXY 20 50 SH 0 PD
REPEAT 120[ FD 1 RT 3 ]
END
```

If you are working only with the Turtle, it is not necessary to TELL TURTLE. Eliminating the position setting in TRIANGLE, SQUARE and CIRCLE enables the Turtle to draw the shapes wherever it happens to be on the screen.

Once the shapes are on the screen, the children can be given instructions about what to do with the helicopter (or Turtle) in relation to those shapes. For example, the children might draw from a stack of cards which instruct them to do any of the following:

- Fly the helicopter (or move the Turtle) **inside** the square.
- Land the helicopter **on top of** the triangle.
- Land the helicopter **between** the square and the triangle.
- Fly the helicopter **around** the circle.

The COMMAND procedure is a model that offers innumerable possibilities. We've used it for EXPLORING directions and the space in relation to shapes. Hopefully, you will think of lots of other ways to use it. Please send us your ideas.

END■

Turtle Fever

by
Rick Billstein

Just as Pac Man™ fever became the rage of elementary and secondary students around the nation, it appears that a new fever has recently struck the educational community. This fever is not dominated by a circular character with a large mouth which devours power capsules when moved around the video screen. Instead, the principal character is a plain looking triangular figure which is simply known as a "turtle." The turtle is not the hero(ine) in a video game, but rather the display creature that resides on the computer screen in the "turtle graphics" portion of Logo.

Logo is a computer language developed over a 12-year period by Seymour Papert and the staff of the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology (MIT). Papert refers to the turtle as "an object to think with." Using very simple commands called "primitives," students can learn to control the turtle by moving it forward, back, right or left. The turtle can leave tracks on the screen or it can move without leaving tracks.

Pac Man™ involves problem solving under time pressure in that points must be scored and enemy goblins must be evaded or destroyed. However, the problem solving involved in "turtle geometry" has no time pressures—that is, if you don't count the pressure from other students waiting to get on the machine.

Logo allows students to pace themselves. The machine responds no matter how fast or how slowly the students work. Students create and discover many things on their own or through working with their classmates. The satisfaction and the enthusiasm that these students have about the computer and about themselves is something that I don't usually see when students finish a course in BASIC. I have felt their excitement when teaching seven Logo classes for teachers and undergraduates and during my observations of elementary students in classes and computer camps.

I'm not alone in my enthusiasm. All one has to do to see evidence of turtle fever is to pick up any computer education journal. There are articles on Logo in most of them and several now have columns on Logo. Some new journals are devoted almost entirely to Logo. Logo user groups are forming all across the country. Sessions on Logo at computer conferences and mathematics conferences are filled to capacity. Universities and colleges have trouble handling the demand for Logo classes. Since I started teaching Logo classes two years ago, the number of requests I receive for materials and information has continually increased. I worry that when I tell people about Logo and about the impact I think it will have on education, they will think, "Wait a minute, aren't you the same guy that told us the metric system was coming?"

I would have trouble defending my position statistically, but for now the feelings I have are enough for me. Logo is good enough to survive in spite of what mistakes we educators may make. I also believe that educators will do more good with Logo and make fewer mistakes if they avoid some misconceptions, are aware of some implementation problems, and have a good grasp of uses of Logo.

More Than Graphics

Logo is powerful because it is more than just turtle graphics. Turtle graphics are very important in the language but the list processing capability of many versions of Logo is also important. Lists are all-purpose data structures that are particularly well-suited for artificial intelligence applications. (For an example of a use of the list processing capability of Logo, see "List Processing in Logo" by Tony Stavely, *The Computing Teacher*, Vol. 10, No. 4.) Whether or not teachers should become proficient with the list processing capability of Logo is another topic, but at least they should be able to follow simple list manipulation routines. My experience is that teachers become attracted to Logo because of turtle graphics and do very little else in the beginning. After a time, many explore the other features of Logo and become very enthused about them. This is true even though list processing is harder to learn than turtle graphics. (This problem has prompted Bob Tinker in an article called "Logo's Limits: Or Which Language Should We Teach?" to begin talking about the need for a "beyond-Logo" language.

Not Just for Kids

Logo is usually advertised as a computer language for children. Logo is not just for kids. It is a powerful, general-purpose programming language that can be used with students at all levels. For example, Professor Andrea di Sessa of MIT uses Logo to teach physics to undergraduate students. He created a "dynaturtle," a dynamic turtle which obeys Newton's Laws of Motion. Working with the dynaturtle helps students better understand how forces affect the motion of objects. The depth of Logo and chess are alike. Elementary students can

learn the basic tactics and pleasures of chess, but the grandmasters are still striving to learn more about the game. The designers of Logo refer to it as "a language with no threshold and no ceiling." For an excellent article which discusses many of the powerful features of Logo, see "Why Logo" by Brian Harvey in the August, 1982 issue of *BYTE* magazine.

Logos, Not Logo

Not all products sold with the Logo name are alike. Different companies have built different capabilities into their products. For example, some products will do only the turtle graphics portion of Logo and no list processing. Some turtle graphics accept only integer inputs while others allow floating point (decimal) entries. Each version of Logo will run on only one brand of computer.

Some, but not all, dialects have "sprite" capability. A sprite can be thought of as an invisible character to which the user can assign attributes. Sprites can carry a shape attribute that the computer already knows or a shape the user designs. In addition, sprites are given color, speed, direction and position. Students can use sprites to form animated graphics. Some Logos such as Texas Instruments Logo have sprite capability built into their product. On other systems, a sprite graphics board and software must be purchased.

Because of the differences in Logos, people are finding that some interesting Logo activities that they would like to try will not work with their set-up. This is not all bad since then we have a new problem-solving experience—understanding the program and translating it into the dialect understood by the user's computer. However, it can be very frustrating to spend a large amount of time typing in a program only to find that some of the symbols in the program are not compatible with the language loaded into the computer.

Implementation Difficulties

Logo cannot be all things to all people. If speed is important, users will find that the present versions of Logo are slow in comparison to BASIC. Also, the current implementations of Logo have little user workspace. For example, when students in a summer course tried to write a program that would print out all of the letters of the alphabet, they ran out of storage before they were finished. In addition, Logo has trouble handling many numerical applications. For example, when students tried to write a program to compute compound interest, they found that they could not compute daily interest because Logo could not handle numbers raised to the 365th power.

The problems just discussed are problems that can be fixed with more advances in hardware. If the advances of the last few years are indicative of the future, we can look forward to newer, more powerful versions of Logo with much greater capability.

Preparation First

Since Logo is so new, most instructors of inservice and preservice classes on Logo have relatively little background on the subject. Thus, in some situations, the power and uses of Logo may not be as evident as they could or should be. In addition, there is a shortage of resource materials, guidebooks, completed research projects, etc. I am beginning to worry if elementary teachers with a weekend workshop from an instructor with little Logo experience will be able to do an adequate job of implementing Logo into their classrooms. Logo classes must not only discuss syntax and capabilities, but also the philosophy and logistics of exposing Logo to others.

Although Logo is advertised as a language in which the users discover things on their own, *teachers must know Logo*. They must know Logo well enough to guide and encourage students. Teachers should not solve problems for the students, but they should know enough Logo to give appropriate hints before the students' frustration leads to unproductive results. Successful problem-solving experiences require problems that are interesting and challenging but not frustrating beyond students' emotional or intellectual capacity.

"Although Logo is advertised as a language in which the users discover things on their own, teachers must know Logo."

Learning problem-solving skills is a process which takes time and a reasonable environment. Students love to work on the machine and they become very engrossed in their own projects. But when students experience shortened class periods, limited machine access, and increasing amounts of course content to cover, the marvelous gains that educators expect from Logo may not be as evident as we would like. Good classroom management and administrative support are critical considerations when implementing Logo. Decisions will have to be made about how much hands-on time each student needs, whether students should work in groups and if so what size groups, and whether to allow students time on the computer at the expense of other activities taking place in the classroom.

Some Things, Not Everything

We must not expect students to learn everything on a topic just from their Logo experiences. For example, there are topics in the mathematics curriculum that cannot (or should not) be Logo driven.

For example, it would be very hard to study concepts such as surface area or volume using Logo. Children not only need to work with objects that can be drawn on a flat screen, but also with three-dimensional real objects. Instruction in this area must not be neglected. Areas where Logo is useful in mathematics include providing an environment for developing mathematical concepts such as variables; the group properties of numbers; the geometrical concepts of length, angle measure, symmetry, similarity; and coordinate systems.

The use of Logo in itself certainly aids the teaching of computer literacy. Students exercise hands-on control of a computer, learning its capabilities and limitations as they learn a structured programming language. Logo seems to be a useful tool for developing logical thinking and problem-solving skills such as the use of top-down or bottom-up strategies.

With All Deliberate Speed

I have been discussing these points about Logo with many people lately. I'm saying, "Yes, I feel that Logo is good, but let's slow down a little."

"Areas where Logo is useful in mathematics include providing an environment for developing mathematical concepts such as variables; the group properties of numbers; the geometrical concepts of length, angle measure, symmetry, similarity; and coordinate systems."

"Everyone else is doing it" is not a good enough reason to immediately run out and buy the first Logo system available. Remember that Logo implementations often do not work as nicely and easily as you read or hear about. Just as in the story about the tortoise (or in this case, turtle) and the hare, it is sometimes better to trudge along with caution rather than bound about without knowing more about a program and what we want to do with it.

First, get acquainted with Logo. Spend some time finding out how implementations differ. Try to understand the power and limitations of Logo. Consider not only what you want to do with Logo right now, but also what you may want to do with it in the future. Set realistic expectations based on your situation. Consider what hardware is available and whether you can afford it. Will purchasing cheaper hardware now really be more expensive in the long run when everything needed for expansion is considered? Who will handle the teaching, and is adequate training feasible? How easy will it be to accept a new role as a Logo teacher? Answer those questions before hardware and software appear in the classroom.

As Molly Watt has written, "As Logo teachers, our roles will cover a diverse, and perhaps uncomfortable range, including demonstrator, teacher/lecturer, teller, time structurer, problem setter, management solver, arbitrator, decision maker, challenger, helper, collaborator, process sharer, question asker, idea extender, observer, documenter, admirer, enjoyer, time providers, technician and model learner." It's great to catch turtle fever, but like Pac Man,™ let's know what we are gobbling up and where the goblins are. Let's do it right.

END■

Rick Billstein, Department of Mathematical Sciences, University of Montana, Missoula, Montana 59812

BIBLIOGRAPHY

Harvey, Brian. "Why Logo," *BYTE*, August, 1982, pp. 163-193.

Stavely, Tony. "List Processing in Logo," *The Computing Teacher*, December 1982, pp. 43-47.

Tinker, Bob. "Logo's Limits: Or Which Language Should We Teach?", *Hands On*, Vol. 6, No. 1, pp. 1-6.

Watt, Molly. "What is Logo," *Creative Computing*, October, 1982, pp. 112-129.

THE LOGO CENTER

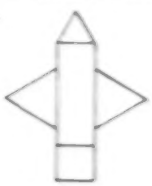
Kathleen Martin • Tim Riordon

Editor's Note: Our column this month features the work of Hillel Weintraub of Doshisha International High School in Japan. Weintraub has started to go beyond turtle graphics, using Logo to do tasks other programmers use BASIC to accomplish. People are beginning to discover all sorts of things you can do with Logo. For example, Dan Weston of Salem, Oregon, has created some procedures that allow text to be printed on the turtle graphics screen. As an overture to the main feature, we also present "Thinking in 'Chunks'."

Kathleen Martin, Dept. of Education, Univ. of Dallas, Dallas, TX 75061. Tim Riordon, 3375 University St., Eugene, OR 97405.

Thinking in "Chunks"

Let's assume that your children can teach the Turtle to make triangles, squares and rectangles, and that they know how to control the size of these shapes. One of your future astronauts brings you the drawing below.



She has figured out that she knows how to teach the Turtle to make all of the pieces of the rocket and, consequently, should be able to teach it TO ROCKET. Let's look at several class activities in which you could engage your children in order to help them to find a way of thinking about this problem.

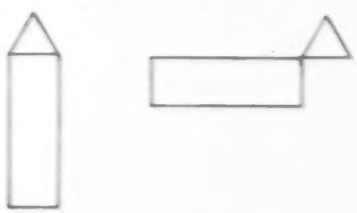
Make some large cardboard cut-outs of the different pieces of the rocket: a long rectangle, a small triangle, two larger triangles and a square. Write the procedure for each of the pieces on the cardboard.

```
TO NOSE (small triangle)
REPEAT 3 [FD 15 RT 120]
END

TO BODY (rectangle)
REPEAT 2 [FD 15 RT 90 FD 50 RT 90]
END

TO TAIL (square)
REPEAT 4 [FD 15 RT 90]
END

TO LEFT.WING (larger triangle)
REPEAT 3 [FD 20 RT 120]
END

TO RIGHT.WING (larger triangle)
REPEAT 3 [FD 20 LT 120]
END
```

Give a "chunk" of the ROCKET to each of five children. Have the other children describe what the Turtle would have to do in order to get the "chunks" of the ROCKET together. This is no easy task since the Turtle needs detailed instructions. Your children could choose to assemble the ROCKET in a number of ways, but let's assume that they start at the top. The top is a small triangle, but the triangle has any number of possibilities, a few of which are shown below:

Discuss with the children how they will get the Turtle to draw the correct triangle. Once the NOSE has been drawn the children have to determine how to connect it with the BODY. This again involves getting the Turtle pointed in the correct direction. The discussion proceeds in this way, emphasizing to the children that the assembly of the ROCKET "chunks" necessitates paying careful attention to the position and heading of the Turtle.

As the children try to figure out how to attach the BODY of the ROCKET to the NOSE, they must attend to another factor. Look at the procedure for the BODY. How is it described? If the Turtle turns so that it is pointing to the right and draws the short side of the rectangle first, the BODY will be vertical. If, however, the Turtle turns so that it is pointing down, the BODY will be horizontal.



Let's assume that the kids have the NOSE and BODY together and now want to add the TAIL.

Where is the Turtle? It's sitting up at the corner where the NOSE and BODY meet and facing to the right, the same direction as it pointed when starting the BODY. Before drawing the TAIL, the Turtle must be moved to the bottom of the ROCKET. This means turning right another 90 degrees and then going forward the length of the rocket BODY. Before making the TAIL, the children must once again study the procedure that draws the TAIL. The procedure starts with a forward move of 15 and then turns right. Look what will happen to the ROCKET:

Ask the children how the situation can be corrected. They will probably see that the simplest solution is to change the TAIL procedure to a left turn.

It might be helpful to write on the blackboard the set of instructions upon which the children agree. If the turns are not exactly 90 degrees, then simply indicate that they are more than or less than 90 degrees. The exact turn can be figured out on the computer.

```
TO ROCKET
RT (less than 90)
NOSE
RT (until facing along the bottom line of the
   NOSE)
BODY
RT 90
FD 50
TAIL (changing RT turn to LT turn)
```

Continue the conversation with the children until the LEFT.WING and RIGHT.WING have been figured out. They are then prepared to go to the computer and try out the ROCKET. They should be able to handle the few changes that they might have to make at this point. One of these changes will be starting the Turtle higher on the screen.

This same project can be approached with a ball of yarn. Before beginning, have the children describe the procedure for each part of the ROCKET and write these on the blackboard. (They will be the same procedures as were previously written on the cardboard geometric shapes.) Select one student to be the Turtle. Give her the ball of yarn with which to leave behind her turtlesteps as she moves through the various commands. Select other children to hold the string at the corners of the geometric shapes that will be drawn. As with the other activity, start with the NOSE. The children will probably suggest immediately that the Turtle turn right before making the small triangle. The "Turtle" child gives the end of the yarn to another child. She then turns right less than 90 degrees in order to face in the appropriate direction. She then repeats three times taking 15 baby steps foward and turning right 120 degrees, ending up where she started. Have the other children describe the turn she must make before drawing the BODY of the ROCKET. The discussion proceeds until the ROCKET is complete.

Depending upon the age group with which you are working, the projects can be simpler or more complex. For example, a simple house could be drawn using a square and a triangle. Or the children could construct a whole city of varying size buildings. The important thing is that the children recognize the power of thinking in "chunks" and the attention that is needed in properly orienting the "chunks" to one another. END■

Putting Logo to Work

by
Hillel Weintraub

When I first began to learn about Logo and to talk to others about this language, one question that kept coming up was, "Yes, those are nice graphics, but what can it really do?" By "really do," I think we all meant, "What can it do that is *useful*," like BASIC programs that could count things or save records or write grades. One of my colleagues at school often asked, "If it's so good, why don't people write programs in it?" and "What else can you use it for besides drawing boxes?" Not knowing how to answer at that point, I became annoyed at his foolish questions. I had something important invested in Logo—my time and energy—and pointing out any weakness was rather like telling me that my sons were indeed not all I thought they were.

Realizing that neither my sons nor Logo are perfect, I admit that Logo has not yet been used for certain applications—however, I am still hesitant to say that it's a weakness in Logo itself. All I can say honestly at this point is, "*I* don't know how to do this in Logo yet."

It is misleading to imply that Logo is a simple language to master. The early steps were easy for me, but once I moved out of the basic graphic primitives, it has taken many hours of reading, playing and experimenting to make added gains in under-

standing. And even then, those gains are sometimes so tentative or nebulous that if I don't practice with them and use them over and over, they are gone the next week and I have to start once more.

This shouldn't be surprising to us—what powerful tools are easy to master? Sure, a hammer is easy to pick up, an electric saw can be plugged in without a fuss; we can learn to use both in a short time. Learning to draw a box with Logo is like learning to hammer a nail—it's a snap. But learning to use nails to build a table or a room or a house—those are much more complex learnings. Just as there are many levels of difficulty in creating a table or house, there are myriad levels of creation for program writing.

It would be helpful for those of us who are still learning to pound nails to have available the kind of books that are plentiful in BASIC—books with fairly short programs that can be copied exactly and run or adapted and run. Such programs provide a good way to expand understanding of the language. Japan is filled with magazines with nothing but short game programs for students to copy. At the present, they are all in BASIC, but it seems clear that they are an attractive way for a lot of people to study.

I also have been pleased to see recent attention being given to non-graphic Logo applications in *The Computing Teacher, The National Logo Exchange* and *The Young People's Logo Association Newsletter*, so that we can begin to gather a body of programs which use Logo to do many different kinds of things.

My program itself grew from an article in the December, 1982 *The Computing Teacher* called "List Processing in Logo" by Tony Stavely. I offer my program to anyone who might be looking for a way to use Logo or to introduce the Logo editor as a word processor. One possible project would be to change the body so that it is an invitation to a school dance, rock concert or birthday party.

It is written in MIT Logo, but has been adapted for LCSI Logo as well. Both programs have been contributed to the YPLA software library and can be ordered from them.

Although the program is quite self-explanatory, I have added comments where I think they may be helpful. Many programs use a SETUP program to start, but to keep everyone on his/her toes, I've used a START program to setup.

```
TO START
  MAKE "INVFLG 232
  PROCEDURE CREDITS
END
```

This procedure is just there to give a convenient name to type to start the operation of the program. For some reason unknown to me, it is necessary to initialize the variable INVFLG with the number 232.

```
TO PROCEDURE CREDITS
  CLEARTEXT
  PRINT [***SETUP FOR APPOINTMENTS***]
  PRINT [H. WEINTRAUB]
  CURSOR 0 3
  HLS 5 PRINT [TATARA, TANABE-CHO, TSUZUKI-GUN]
  HLS 8 PRINT [KYOTO-FU, JAPAN 610-03]
  PRESS.RETURN.TO.CONTINUE
  SETUP
END
```

Originally this was written with a PAUSE or WAIT command that was on a timed cycle and would automatically go to SETUP after a specified time. Under the belief that the user should be able to control the pace of the material being presented, I worked on a PRESS.RETURN.TO.CONTINUE procedure which should be useful in any program where you want the user to decide when the next procedure should be called.

```
TO HLS :NUM
  REPEAT :NUM [PRINT1 [ ]]
END
```

This little procedure was written as a way to make it easy to do screen editing. The cursor command in Logo is based on an absolute position, HLS and VLS are based on the position of the cursor when the command is given.

```
TO PRESS.RETURN.TO.CONTINUE
  CURSOR 5 21
  PRINT1 SENTENCE [PRESS] [ ]
  INVERSE FLASHING
  PRINT1 ["RETURN"]
  NORMAL
  PRINT SENTENCE [ ] [TO CONTINUE]
  MAKE "ANSWER READCHARACTER
  TEST :ANSWER = CHAR 13
  IFFALSE PRINT [NO, THE RETURN KEY!] PRESS.RETURN.TO.CONTINUE
END
```

I was really pleased about being able to write this procedure because it is a standard subroutine in BASIC. It took me a long time to get it operating, but I learned a lot in the process of working it out. Originally I tried to have it as a PRESS.SPACE.BAR.TO.CONTINUE routine, but I was not able to make it work with the ASCII character 32 for SPACE, rather than ASCII character 13 for RETURN.

```
TO SETUP
  CLEARTEXT
  PRINT [***SETUP FOR STUDENT APPOINTMENT***]
  VLS 2
  PRINT [WHAT DATE AT TOP OF LETTER?]
  CURSOR 14 7
  MAKE "HEAD.DATE REQUEST
  VLS 1
  PRINT [WHERE DO YOU WANT TO MEET THE STUDENT?]
  VLS 1 HLS 7
  PRINT1 [IN THE ----------]
  CURSOR 14 11
  MAKE "PLACE.OF.MEETING REQUEST
  VLS 2
  PRINT [HOW WILL YOU SIGN YOUR NOTE?]
  CURSOR 14 16
  MAKE "TEACHER'S.NAME REQUEST
  SETUP1
END
```

This procedure sets up the variables which don't need to be changed for each student letter, but need to be set at the beginning of the program for all the letters. If the user wishes to change either the date at the top of the letter, the meeting place or the signature, CTRL G, and then START would need to be typed again.

Your formatting tastes may be different from mine, so the CURSOR, HLS or VLS procedures may need to be changed.

```
TO VLS :NUM
  REPEAT :NUM [PRINT []]
END
```

Like the HLS procedure, this is a way for me to add spaces and ease my screen/printer formatting.

```
TO SETUP1
  VLS 2 PRINT1 [STUDENT'S NAME:]
  MAKE "STUDENT'S.NAME REQUEST
  HLS 6 PRINT1 [TIME OF MEETING:]
  MAKE "TIME.OF.MEETING REQUEST
  HLS 12 PRINT1 [DATE OF MEETING:]
  MAKE "DATE.OF.MEETING REQUEST
  PRINT.LETTER
  PRESS.RETURN.TO.CONTINUE
  CLEARTEXT
  SETUP1
END
```

This procedure is used to enter the variables which need to be changed for each letter or note.

Once all the information is entered, the letter is then printed on screen and printer. Then the next note's information is asked for. This will continue until the user stops the program by typing CTRL-G.

```
TO PRINT.LETTER
  OUTDEV 3
  PRINT1 CHAR 27 PRINT1 CHAR 64
  PRINT1 CHAR 27 PRINT1 CHAR 69
  PRINT1 CHAR 27 PRINT1 CHAR 71
  VLS 2
  LETTER
  VLS 2
  OUTDEV 0
END
```

This procedure relates to turning the printer on and sending certain codes to the printer, in this case an Epson printer, in order to get the kind of print which is strongest and clearest. Working out the codes for your particular printer will take time, but once they are known, they will be most useful.

```
TO LETTER
  HEAD
  BODY1
  BODY2
END
```

This procedure simply calls the parts of the letter.

```
TO HEAD
  CLEARTEXT
  HLS 22 PRINT :HEAD.DATE
  VLS 1
  PRINT1 SENTENCE [DEAR] :STUDENT'S.NAME
  PRINT [,]
  VLS 1
END
```

This procedure sets up the heading, including the date the note was written (entered in the SETUP procedure) and the student's name (entered in the SETUP1 procedure).

```
TO BODY1
  PRINT SENTENCE [LET'S GET TOGETHER IN THE] :PLACE.OF.MEETING
  VLS 1 HLS 7 PRINT [AT THIS TIME:]
  HLS 14 PRINT :TIME.OF.MEETING
  VLS 1 HLS 7 PRINT [ON THIS DATE:]
  HLS 14 PRINT :DATE.OF.MEETING
  VLS 1
END
```

This procedure uses the three variables entered in SETUP1.

This formatting took a lot of time because I wanted the output to look neat on both my screen and on the printout. However, it still needs work, and your setup may require some changes.

```
TO BODY2
  PRINT [I WOULD LIKE TO EXCHANGE IDEAS WITH YOU]
  CURSOR 0 13
  PRINT [ABOUT OUR WORK IN LOGO SINCE WE TALKED]
  PRINT [LAST TIME. BE SURE AND LET ME KNOW IF]
  PRINT [YOU CAN'T MAKE IT AND WE'LL SET ANOTHER]
  CURSOR 0 16
  PRINT [TIME. I'M LOOKING FORWARD TO OUR TALK.]
  VLS 2 HLS 22
  PRINT : TEACHER'S.NAME
END
```

These cursor commands are there to keep the screen from having an unwanted line space due to the Logo Editor's system of automatically sending a return command when the last space on a line is filled.

```
TO NORMAL
  DEPOSIT :INVFLG 255
END

TO INVERSE
  DEPOSIT :INVFLG 0
END

TO FLASHING
  DEPOSIT :INVFLG 64
END
```

The NORMAL, INVERSE and FLASHING procedures are available on the MIT utility disk.

END■

Hillel Weintraub, Acting President, S.M.I.L.E., Doshisha International High School, Tatara, Tanabe-cho, Tsuzuki-gun, Kyoto-fu, Japan 610-03.

Coming Up

We need some reader input to help us with a column we are planning later this year: Logo and Numbers. Can Logo be used as easily as BASIC to do the kinds of programming students do in middle and high school math classes? Can you contribute any ideas you have relevant to this topic?

THE LOGO CENTER

Kathleen Martin • Tim Riordon

Non-Graphics Logo Applications

"Logo is just for drawing graphics, right?" How often have we heard that? This month our column will focus exclusively upon some fun, non-graphics applications. Your students might enjoy getting involved in making cryptograms and codes. Or what about having them explore the concept of a musical code or a musical language? The computer will sound like C3PO. Have your students recall the idea of a musical language used in *Close Encounters of the Third Kind.*

Cryptograms

We present here a set of procedures for Terrapin Logo which will take a word typed in by the player and turn it into a cryptogram. For example, if you tell the computer to encode "Rumpelstiltskin," Logo might print out

HLRTAQZDNQDZENB

We have three purposes in presenting these procedures. First, kids will enjoy using them even if they don't work with the procedures themselves or try to modify them. Second, these procedures almost beg for two further modifications, making it a starting point for a student project. Finally, the way the procedures work illustrates an interesting way of storing and accessing information in Logo. (This approach is also used in the next section on musical codes.)

The superprocedure that runs the whole "code show" is TO CODE. It does three jobs: Creates a code book in which every letter of the alphabet has been assigned a secret counterpart (the code letter); finds out what word the player wants to encode; and prints out the encoded message that the user typed in.

```
TO CODE
 CLEARTEXT
 PRINT [WHAT WORD DO YOU WANT TO ENCODE?]
 PRINT []
 PRINT [PLEASE BE PATIENT...]
 PRINT [I NEED 30 SECONDS TO MAKE UP]
 PRINT [THE SECRET CODE!]
 CREATE.CODEBOOK
 PRINT []
 MAKE "MESSAGE FIRST REQUEST
 PRINT [] PRINT []
 PRINT SENTENCE [HERE IS THE ENCODED VERSION OF] :MESSAGE
 PRINT [] PRINT []
 PRINT ENCODE :MESSAGE
END
```

Before looking at the four procedures which create the code book, consider two ways that the information might be stored and accessed. One way is to create a list of letter pairs—the first letter would be the letter in

question; the second would be the code letter that is to be substituted. Such a list (perhaps named "CODE.TABLE) would look like this:

[[A H] [B R] [C E] [D Z] [E M] [F B] etc.]

Using this code, the word FADE would be translated as BHZM. If this approach were used, the ENCODING would involve some sort of LOOKUP procedure:

LOOKUP "F :CODE.TABLE would output the code letter "B
LOOKUP "A :CODE.TABLE would output the letter "H
LOOKUP "D :CODE.TABLE would output the letter "Z
LOOKUP "E :CODE.TABLE would output the letter "M

The letters "F, "A, "D and "E are called "keys." This approach, looking up information in a table according to "keys," works okay for a short table. But for a long table—like the entire alphabet—this approach will take a long time. Here's why. The CODE.TABLE is a list of 26 items. (Each item is a letter pair—the key letter and the code letter.) Every time a key is searched for, the search begins at one end of the list and each item is examined until the "key" is found.

The alternative used in our cryptogram program is to create a set of 26 global variables. These might have been created directly by typing in

MAKE "A H MAKE "B R MAKE "C E MAKE "D Z . . . MAKE "Z Q

The procedure CREATOR :ASCII.OF.LETTER does all of this automatically.

```
TO CREATOR :ASCII.OF.LETTER
 IF :ASCII.OF.LETTER > 90 STOP
 MAKE CHAR :ASCII.OF.LETTER CHAR GENERATOR
 CREATOR :ASCII.OF.LETTER + 1
END
```

CREATOR begins with the letter whose ASCII number is 65 (i.e., the letter A) and recurs until it has worked with the letter whose ASCII number is 90 (the letter Z.) The line in CREATOR beginning with MAKE takes a letter of the alphabet as a name for a global variable. It assigns to this global variable some other letter—the code letter. The procedure GENERATOR does two things. It makes certain that the assignments are made randomly. It also tests to see that the same code letter isn't assigned to two different alphabet letters.

```
TO GENERATOR
 MAKE "NUM 65 + RANDOM 26
 TEST NOT MEMBER? :NUM :NUMLIST
 IFTRUE MAKE "NUMLIST LPUT :NUM :NUMLIST
 IFTRUE OUTPUT :NUM
 IFFALSE MAKE "NUM GENERATOR
 IFFALSE OUTPUT :NUM
END
```

Let's see what happens when the input number to CREATOR reaches 75.

	Comments
CREATOR 75 IF 75 > 90 STOP	The procedure will not stop.
MAKE CHAR 75 CHAR GENERATOR	CHAR 75 is the letter K. So, this line is like typing MAKE "K . . .
CREATOR 75 + 1 END	What gets associated with the global variable "K is a code letter whose ASCII number is produced by the procedure GENERATOR. If GENERATOR output the number 67, then the letter C would be the code letter for the variable named "K.

The procedure GENERATOR has a simple job. It is supposed to generate random numbers from 65 to 90. It outputs this number provided that the number has not already been output before. So, each time GENERATOR is called upon to output a number, it does each of the following tasks:

Creates a random number (:NUM) from 65 to 90.

TESTS to see if (:NUM) is not a member of the list of numbers (:NUMLIST) that have already been generated.

If :NUM has not been used already, then :NUM is output. It is also added to the ever-growing list of numbers that GENERATOR has already used.

On the other hand, if :NUM is already on the :NUMLIST, then :NUM must take on another random value before it is output.

Here are the other two procedures needed to create the code book:

```
                                                    Comments
TO CREATE.CODEBOOK                                  This begins the code-making process.
 MAKE "NUMLIST []
 CREATOR 65
END

TO MEMBER? :WORD :LIST                              This subprocedure is used in GENERATOR.
 IF :LIST = [] OUTPUT "FALSE
 IF :WORD = ( FIRST :LIST ) OUTPUT "TRUE
 OUTPUT MEMBER? :WORD BUTFIRST :LIST
END
```

Remember that the four code-making procedures will create 26 global variables named A, B, C, etc. The value of each of these variables will be some randomly selected code letter. If you were to type, for example,

```
PRINT :A
```

the computer would print H (the code letter for A).

The final part of the superprocedure TO CODE has the job of printing out the encoded message. Two short procedures do this job.

```
                                                    Comments
TO ENCODE :MESSAGE                                  See #3 following.
 IF :MESSAGE = " OUTPUT "
 OUTPUT WORD GET.CODE.LETTER FIRST :MESSAGE ENCODE BUTFIRST :MESSAGE
END

TO GET.CODE.LETTER :LETTER
 OUTPUT THING :LETTER
END
```

GET.CODE.LETTER has, as its input, a letter from the message being encoded. For example, GET.CODE.LETTER "A would output A's code letter—perhaps H.

ENCODE uses a strategy which is frequently used in Logo programming. We'll explain it using the following example:

1. If CAB is being encoded, then the job is done when E—the code letter for C—is attached to the result of encoding AB.
2. The job of encoding AB is done when H—the code letter for A—is attached to the result of encoding B.
3. The job of encoding B is done when K—the code letter for B—is attached to nothing! At this point, the process should STOP. The first line in the procedure— IF :MESSAGE = " OUTPUT " —does this job.

4. Summing up: K gets attached to nothing, H gets attached to K, and then E gets attached to KH. The result is EHK.

After your students work with CODE a few times, they will probably want to make two modifications. As the procedure now works, a new code is generated each time someone plays. This is time consuming. The first modification would be to reorganize things so that a player only needs to create the code once, and can then use the same code to make many plays. Help your students see that a new procedure—TO PLAY—is needed. PLAY will find out what word to encode, and then print it out. If TO PLAY is recursive, the game can go on with a minimum of typing.

The second modification is to redesign the procedures so they will encode messages longer than one word. For help on how to do this, your students should see how the same problem is solved in the next section: Musical Language Code.

Musical Language Code

This project was inspired by listening to C3PO and the thought that there might be galactic civilizations with musical languages. These procedures work with TERRAPIN Logo. They may work with Krell's Logo also. We would appreciate it if someone would send us this project in other Logo versions.

This project began by making procedures that would musically encode single words. We present here the slightly more elaborate version which will encode whole messages. The more elaborate version is given to help those who want to work on the modification suggested above for Cryptograms.

The superprocedure that begins the process is CODEM (M for Music!). It does two jobs: it sets up the musical code and calls on a recursive procedure that will continue to ask players what messages they wish to encode. Here's CODEM.

```
TO CODEM
 PRINT [MAKE SURE YOU HAVE A UTILITY DISK IN]
 PRINT []
 PRINT [THEN PRESS A KEY TO CONTINUE]
 MAKE "KEYPRESSED RC
 CLEARTEXT
 SETUP
 MUSICMAKER
END
```

SETUP, with the help of two subprocedures, creates the musical code book. As with the Cryptogram project, the code is stored in 26 global variables. Each letter is the name of a variable. Its associated value is a number which can be used in a procedure TO PLAY.NOTE to make a musical tone. Here are the three procedures with comments.

Comments

```
TO SETUP
 DUS [BLOAD MUSIC.BIN]
 MAKE "FUDGE 2
 MAKE "BASE.PERIOD 140
 MAKE "NOTELIST []
 MAKE "TONE ( .EXAMINE 43634 ) + 256 * .EXAMINE 43635
 WHOLE.CODE
END
```

MUSIC.BIN is Terrapin's magic music file. I don't understand the first five lines of this SETUP procedure either. But it does the job!

```
TO WHOLE.CODE
 MAKE.CODE 65 20
 MAKE "# 16384
END
```

WHOLE.CODE includes all the letters and a musical rest, denoted by #. The rest will be used to pause between words of a message.

```
TO MAKE.CODE :ASCII.OF.LETTER :PERIOD
 IF :ASCII.OF.LETTER > 90 STOP
 MAKE CHAR :ASCII.OF.LETTER :PERIOD
 MAKE.CODE :ASCII.OF.LETTER + 1 :PERIOD *
 1.05945
END
```

This procedure creates 26 global variables, beginning with A, whose ASCII is 65, and ending with the letter whose ASCII number is 90. This procedure is simpler than the one used in the Cryptogram project—we did not care about randomness. It's fun to fool around with the multiplier 1.05945. As it is now, you will get standard music pitches. Try changing it to create a pentatonic scale!

Once the code has been created, MUSICMAKER begins running. It does five things over and over:

1. Asks what message (:MESSAGE) to encode.
2. Creates a list of numbers (:NOTELIST) which will later be fed into the procedure PLAY.NOTE to produce the sound. :NOTELIST will be a list of numbers like

   [59.9174 63.4795 59.9174 16384 16384]

   These numbers, fed into PLAY.NOTE, will make the music for TUT, followed by a rest.
3. Plays the notes.
4. Clears out the :NOTELIST.
5. Runs another copy of itself so a player can type in a new message.

Here are MUSICMAKER and the two procedures that create the list of numbers called :NOTELIST.

```
TO MUSICMAKER
 PRINT [WHAT MESSAGE WOULD YOU LIKE ENCODED]
 PRINT [INTO THE LANGUAGE OF MUSIC?]
 PRINT [] PRINT []
 MAKE "MESSAGE REQUEST
 PRINT [PRESS A KEY TO HEAR HOW IT]
 PRINT [SOUNDS IN OUR MUSICAL LANGUAGE]
 MAKE "KEYPRESSED RC
 CREATE.NOTELIST :MESSAGE
 TUNE :NOTELIST
 CLEARTEXT
 MAKE "NOTELIST []
 MUSICMAKER
END

TO MENCODE :ITEM
 IF :ITEM = " OUTPUT [16384 16384]
 OUTPUT SENTENCE THING FIRST :ITEM MENCODE BUTFIRST :ITEM
END

TO CREATE.NOTELIST :MESSAGE
 IF :MESSAGE = [] STOP
 MAKE "NOTELIST SENTENCE :NOTELIST MENCODE FIRST :MESSAGE
 CREATE.NOTELIST BUTFIRST :MESSAGE
END
```

Once you have found the lines in MUSICMAKER that do the five jobs mentioned above, let's focus on the problem of creating the list of notes.

MENCODE will take a single word, refer to the global variable named by each letter in the word, and create a list. The list will be the values—the numbers like 59.9174 for "T. The list will always end with the two numbers for a musical rest—16384 16384. If this pause between words is not long enough, then you could change the first line of MENCODE so that 3 or 4 rest numbers were tacked onto the end of each list.

So, MENCODE will produce the musical code for different words in a message—but only one at a time. CREATE.NOTELIST is the procedure that causes a long list of numbers to be generated. If you study this procedure, it will suggest a way to modify the Cryptogram program so that it will accommodate messages of more than one word.

Finally, here are the three procedures that will take the :NOTELIST and cause a tune to be played.

```
TO TUNE :NOTELIST
 IF :NOTELIST = [] STOP
 PLAY.NOTE FIRST :NOTELIST 40
 TUNE BUTFIRST :NOTELIST
END
```

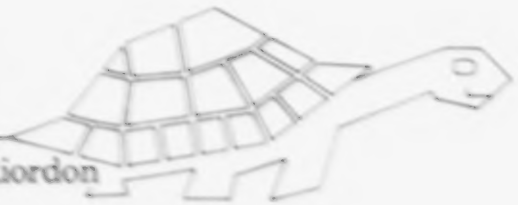
```
TO .CALL.2 :ADDR :INPUT1 :INPUT2
 .CALL :ADDR ( ROUND :INPUT2 ) + ( ROUND :INPUT1 ) * 65536
END

TO PLAY.NOTE :PERIOD :DURATION
 .CALL.2 :TONE :PERIOD - :FUDGE :DURATION * :BASE.PERIOD / :PERIOD
END
```

What's left for students to do with these procedures? Well, how about creating a MORSE CODE program? The player types in the message, and out comes the sound of MORSE code! END■

[Tim Riordon, 3375 University, Eugene, OR 97405. Kathleen Martin, Dept. of Education, Univ. of Dallas, Dallas, TX 75061.]

THE LOGO CENTER

Kathleen Martin • Tim Riordon



Big Kids Helping Little Kids

This month our focus is on helping teachers of primary age students with Logo. Some primary teachers express a reluctance about having students work with Logo—not because they believe the activity is unworthy, but because they foresee classroom management problems. How can I break away from the rest of my class to help kids do Logo at the computer? How can kids who don't read work with all those typewriter keys? Many primary teachers are overlooking a readily available resource of classroom volunteers who have an extensive knowledge of Logo, who are available every day, who will work for no pay. These volunteers have a further qualification—they understand how to explain things in ways that other children understand. In order for your primary teachers to tap into this volunteer service, have them contact the fifth grade teacher in your school whose students have been doing Logo for a year!

Last year in Missoula, Montana, Shirley Torgerson moved their school's computer into the kindergarten teacher's classroom for the last month of school. Every half hour, one of her students would leave the classroom and go to the kindergarten. Each student would work on a one-to-one basis with a kindergarten student, teaching Logo. Kindergarten students would spend 15 minutes with their fifth grade tutor, so each tutor worked with two kindergarten students.

Shirley's fifth grade students had been organized as partners throughout the school year. This activity also involved partnerships, though they had an unusual purpose.

"Choose a partner you do not wish to work with," was Shirley's direction now. You see, the partner's responsibility was to take notes, record assignments and turn in assignments for the other partner when s/he was working in the kindergarten room. This system had an added payoff when kids were sick and parents called in for make-up work.

What and how does a fifth grade tutor teach a kindergarten student about Logo? There are some procedures tutors can create that will have their young students use some basic shapes and some designs that can be made with the shapes. You'll see ways to add to these procedures. The time will come when the primary students will want to draw some designs of their own.

Sometimes it is good to encourage older children to write Logo programs that younger children can use. The older child (perhaps a fifth grader) becomes a mentor of sorts to the younger child (a first grader or maybe a kindergartener). As a consequence of the interaction, the older child usually discovers modifications s/he wants to make in his or her program, or even conceives of an entirely new program. In extending the programs to others, s/he learns a lot about how to structure programs more effectively.

> ***"Many primary teachers are overlooking a readily available resource of classroom volunteers who have an extensive knowledge of Logo, who are available every day, who will work for no pay. These volunteers have a further qualification—they understand how to explain things in ways that other children understand."***

There is no end to the kinds of programs that can be developed, but you might want to start your students on a program that combines some simple list processing with turtle graphics. The program that we will describe is called DESIGN. It allows the student to select a basic shape and then proceeds to use the shape in making a more complex design. The choice is limited to three shapes (square, triangle and circle), but can be easily changed to include more. The program is rudimentary and lends itself to a variety of modifications. It is written for the TI computer, but does not differ significantly from other versions of Logo.

```
TO DESIGN
  CS TELL TURTLE
  PRINT [WHAT SHAPE WOULD YOU LIKE]
  PRINT [TO SEE?]
  MAKE "ANSWER READWORD
  IF :ANSWER = "SQUARE SQUARE
  IF :ANSWER = "TRIANGLE TRIANGLE
  IF :ANSWER = "CIRCLE CIRCLE ELSE PRINT [TRY
    ANOTHER SHAPE.] WAIT 150
  DESIGN
  END

TO READWORD
  OUTPUT FIRST READLINE
  END
```

READWORD is a command that is already defined in some versions of Logo, e.g., Apple Logo. Defining it for TI Logo is simple enough. The command MAKE awaits the input of the student (in this case square, triangle or circle), and then returns to the line of the program where that input is executed. If a shape other than the three defined is chosen, the student is instructed to "try another shape." If SQUARE is chosen, the following subprogram describes what will happen.

```
TO SQUARE
  SQ WAIT 100
  PRINT [DO YOU WANT TO SEE A DESIGN]
  PRINT [MADE FROM THE SQUARE?]
  TEST FIRST READWORD = "Y
  IFT CS PRINT [OK, HERE GOES.] SQUARES WAIT
    150
  END
```

A square is drawn and then the student is asked if s/he wants a design drawn using the square as the basis for the design. If the student answers yes, the design will be drawn—if no, the program returns to the original question. The design is drawn with squares defined by the following two subprograms.

```
TO SQUARES
  REPEAT 18 [SQ RT 20]
  END

TO SQ
  REPEAT 4 [FD 30 RT 90]
```

Can you visualize the design? The subprograms for TRIANGLE and CIRCLE are structurally the same.

```
TO TRIANGLE
  TRI 30 WAIT 100
  PRINT [DO YOU WANT TO SEE A DESIGN]
  PRINT [MADE FROM THE TRIANGLE?]
  TEST FIRST READWORD = "Y
  IFT [CS PRINT [AROUND AND AROUND] | TRIANGLES
    WAIT 150
  END

TO TRIANGLES
  REPEAT 4 [TRI 20 TRI 30 TRI 40 RT 90]
  END
```

```
TO TRI :N
  REPEAT 3 [FD :N RT 120]
  END

TO CIRCLE
  CIR 5 WAIT 100
  PRINT [DO YOU WANT TO SEE A DESIGN]
  PRINT [MADE FROM A CIRCLE?]
  TEST FIRST READWORD = "Y
  IFT CS PRINT [AROUND AND AROUND WE GO.]
    CIRCLES WAIT 150
  END

TO CIRCLES
  CIR 5
  CIR 4
  CIR 3
  CIR 2
  END

TO CIR :D
  REPEAT 36 [FD :D RT 10]
  END
```

Students can be encouraged to add other shapes to the program. The shapes can be kept simple, like a rectangle, or made more difficult, like a star. The designs, too, can reflect different levels of complexity, depending upon the sophistication of the students.

An off-computer activity that can accompany this project requires pencil, paper and cardboard cutouts of different geometric shapes. Have each student select a single shape that s/he wants to use for a design. Remind students that their pencil is like the turtle, and that they should pay attention to how they are drawing as well as to what they are drawing. This process of getting students to think about and describe the movements of the pencil prepares them to work with Turtle Graphics. Their attention is directed to the same considerations of distance and angle.

The project can be used to motivate the younger children to learn Turtle Graphics. They will want to teach the computer how to make the shapes and designs themselves, rather than simply watching them being made.

```
TO COMMAND
  MAKE "COM READCHARACTER
  IF :COM = "F FD 5
  IF :COM = "R RT 15
  IF :COM = "L LT 15
  IF :COM = "P PENUP
  IF :COM = "D PENDOWN
  IF :COM = "B BK 5
  IF :COM = "N DRAW
END

TO INSTANT
  COMMAND
  INSTANT
END
```

How can your students save a picture they have made using INSTANT?

First, stop INSTANT by pressing < CONTROL > G.

Next, put in a "saving disk" and type FULLSCREEN SAVEPICT "BILL SPLITSCREEN (of course, Bill could be any other name).

The Logo teacher can tell you how to print these pictures on paper so your students will have something to take home to show their parents.

Eventually you will want to add new single keystroke commands to your COMMAND procedure. One useful command is one that will erase lines backwards. If your version of Logo doesn't have PENERASE, then define it as a procedure. Here is the new line you add to COMMAND:

```
IF :COM = "E PENERASE
```

If your Logo doesn't have PENERASE, define it as follows:

```
TO PENERASE
  MAKE "ORIGINAL.COLOR LAST TURTLESTATE
  MAKE "ERASING.COLOR LAST BUTLAST
    TURTLESTATE
  PENCOLOR ERASING.COLOR
  BK 5
  PENCOLOR ORIGINAL.COLOR
END
```

After your students have learned to draw with this version of INSTANT, you can add some more commands. The following four procedures will let your primary students draw with color. Then you can add another command to COMMAND:

```
IF :COM = "C CHOOSE.COLOR PRINT PRINT
```

```
TO CHOOSE.COLOR
  PRINT [PRESS THE COLOR KEY YOU WANT]
  MAKE "COLOR READCHARACTER
  TEST ( ANYOF :COLOR = 1 :COLOR = 2 :COLOR =
    3 :COLOR = 4 :COLOR = 5)
  IFTRUE CLEARTEXT
  IFTRUE PENCOLOR :COLOR
  IFTRUE PRINT [YOU CAN DRAW NOW IF YOUR
    PEN IS DOWN]
  IFTRUE WAIT 200 CLEARTEXT
  IFTRUE STOP
  IFFALSE PRINT [ ] INVERSE PRINT [PRESS ONLY
    THE COLOR BUTTONS] NORMAL PRINT [ ]
  CHOOSE.COLOR
END

TO NORMAL
  DEPOSIT :INVFLG 255
END

TO INVERSE
  DEPOSIT :INVFLG 0
END

TO WAIT :COUNT
  TEST :COUNT = 0
  IFTRUE STOP
  WAIT :COUNT - 1
END
```

When you add these procedures and the new line to command, you will also need to add a line to INSTANT. Here's the new INSTANT:

```
TO INSTANT
  MAKE "INVFLG 232
  COMMAND
  INSTANT
END
```

There are many other commands you can add to the COMMAND procedure. You could make keys like S, C and T cause the turtle to draw a SQUARE, a CIRCLE or a TRIANGLE. Or you could have keystrokes that could cause the turtle to draw things such as SUN, TREE, MOUNTAIN, STREET, HOUSE, etc.

We would like to hear from older students who work with younger students. Write us and tell us ideas that work for you. FORWARD 100 ENDO

(Tim Riordan, 3373 University, Eugene, OR 97403. Kathleen Martin, Dept. of Education, Univ. of Dallas, Dallas, TX 75061.)